Scalable Algorithms and Persistent Data Structures using Geometric Techniques

Rolf Svenning

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Scalable Algorithms and Persistent Data Structures using Geometric Techniques

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Rolf Svenning
October 4, 2025

Abstract

This thesis presents advances in theoretical computer science by developing scalable algorithms and data structures engineered for high performance in the external memory and parallel computation models. The relevance of these models is driven by the unprecedented scale of data generation in modern science, industry, and daily life, where data throughput and parallelism have become paramount to system performance.

We first introduce a novel priority queue for the external memory model that achieves asymptotically optimal insertion complexity with respect to both I/Os and comparisons. This structure is particularly useful for applications characterized by insert-heavy workloads or those where computations conclude with a non-empty priority queue.

Second, we achieve a significant breakthrough on a long-standing open problem by designing the first fully persistent search tree in external memory that matches the asymptotic performance bounds of the classical B-tree. Persistent data structures preserve their history and enable queries and updates on any prior version of the data. Our approach implements a geometric two-dimensional view of persistence by constructing specialized external memory data structures engineered for efficient maintenance and querying that view. We also design more efficient, partially persistent variants that leverage buffering techniques while achieving worst-case I/O bounds. Separately, we adapt a general transformation for persistence to the functional model.

Third, we advance both the theoretical and practical understanding of the all nearest smaller values problem. We prove that a simple heuristic algorithm is, in fact, work-efficient, thereby matching the complexity of a known, theoretically optimal algorithm. Furthermore, we present the first implementation and empirical evaluation of the theoretically optimal algorithm, demonstrating its practical viability. In computational geometry, we resolve the previously open contiguous art gallery problem by presenting a polynomial-time algorithm. Our algorithm uses a greedy strategy to optimally guard the boundary of a polygon using contiguous polygonal chains. Additionally, we introduce a fast convex hull peeling algorithm for two-dimensional outlier detection, which identifies outliers as points whose removal causes a significant decrease in the area of the convex hull.

Resumé

Denne afhandling præsenterer fremskridt inden for teoretisk datalogi gennem udviklingen af skalerbare algoritmer og datastrukturer, optimeret til høj ydeevne i modeller for ekstern hukommelse og parallel beregning. Relevansen af disse modeller er drevet af det hidtil usete omfang af datagenerering i moderne videnskab, industri og dagligliv, hvor datagennemstrømning og parallelisme er blevet altafgørende for systemers ydeevne.

Først introducerer vi en ny prioritetskø til modellen for ekstern hukommelse, der opnår asymptotisk optimal indsættelseskompleksitet, for både I/O'er og sammenligninger. Denne struktur er særligt anvendelig for applikationer karakteriseret ved indsættelsestunge arbejdsbyrder, eller hvor beregninger afsluttes med en ikke-tom prioritetskø.

Dernæst opnår vi et markant gennembrud på et længe åbent problem ved at designe det første fuldt persistente søgetræ i ekstern hukommelse, der matcher de asymptotiske ydeevnegrænser for det klassiske B-træ. Persistente datastrukturer bevarer deres historik og muliggør forespørgsler samt opdateringer på enhver tidligere version af dataene. Vores tilgang implementerer en geometrisk, todimensionel forståelse af persistens ved at konstruere specialiserede datastrukturer i ekstern hukommelse, som er udviklet til effektiv vedligeholdelse og forespørgsel på denne repræsentation. Vi designer desuden mere effektive, delvist persistente varianter, der udnytter bufferingsteknikker, og som samtidig opnår worst-case I/O-grænser. Ydermere tilpasser vi en generel transformation for persistens til den funktionelle model.

For det tredje udvider vi den teoretiske og praktiske forståelse af ANSV-problemet (All Nearest Smaller Values). Vi beviser, at en simpel heuristisk algoritme faktisk er work-efficient og derved matcher kompleksiteten af en kendt, teoretisk optimal algoritme. Derudover præsenterer vi den første implementering og empiriske evaluering af den teoretisk optimale algoritme og demonstrerer dens praktiske anvendelighed. Inden for beregningsgeometri løser vi det hidtil åbne 'contiguous art gallery'-problem ved at præsentere en polynomiel-tids-algoritme. Vores algoritme anvender en grådig strategi til at dække en polygons kant optimalt med sammenhængende polygonkæder. Derudover introducerer vi en hurtig 'convex hull peeling'-algoritme til to-dimensionel outlier-detektion, som identificerer outliers som punkter, hvis fjernelse forårsager et markant fald i det konvekse hylsters areal.

Acknowledgments

I am profoundly grateful to the many individuals who have made this journey an experience I will cherish forever.

First and foremost, I extend my deepest gratitude to my supervisor, Gerth. Knowing that he would be my advisor was a decisive factor in my decision to pursue this PhD. His invaluable mentorship has been defined by his genuine care for his students, his trust in our independence, and his consistently open door. He was always generous with his time, whether the conversation was about a difficult research problem or simply a casual chat. I also thank Kasper for his leadership and for cultivating a friendly and supportive research environment.

My research visits abroad were a defining experience of my PhD, and I am deeply grateful to my hosts. I thank Nodari for being an exceptional host at the University of Hawaii at Mānoa, where he introduced me to the world of parallel algorithms. His hospitality went far beyond any expectation, and I am especially grateful for his friendship. My thanks also go to Mike for graciously hosting my extended visit to his fantastic research group at the University of California, Irvine.

I thank my collaborators and friends Casper and Jens Kristian for being a reliable source of support and for the many memorable conferences we attended together. I will remember the academic marathons leading up to submission deadlines, which we somehow turned into something both challenging and fun. My gratitude extends to my co-authors, collaborators, officemates, colleagues, my study group throughout my bachelor's and master's studies, travel companions, climbing and surfing partners, and many others, with special thanks to Andrew, Mads, and Simon.

On a personal note, I owe everything to my family. I thank my parents, Else and Jens-Christian, for their unwavering support, no matter which path in life I chose to pursue. I am especially grateful for the love of science and nature they instilled in me from a young age. To my sister, Liv, thank you for countless conversations during our spontaneous runs, swims, and coffee breaks, when we both needed breathing room during a busy day. To my brother, Asger, thank you for endlessly fascinating discussions that have broadened my perspective, blending computer science with biology, and directly inspiring one of the articles in this thesis.

Rolf Svenning,
Aarhus, July 31st, 2025.

Contents

Part I

Overview

Chapter 1

Introduction

Scalability is, for good reason, a central concept in computer science. Today, technology and digitization are critical and ubiquitous parts of society, leading to the daily generation of vast and rapidly growing amounts of data from a multitude of sources. These include video streaming services, social media platforms, financial markets, large-scale scientific experiments, earth observation systems, and hospitals producing high-resolution medical imaging, among many others. Not only is storing and analyzing massive volumes of data a central challenge across a wide range of applications, but the effectiveness and sophistication of these applications also often increase with access to even more data. This escalating demand has driven computer science to explore new computational models capable of handling data sizes that vastly exceed the capacity of a single machine. Some key developments in this direction include distributed computing for scaling across multiple machines, streaming algorithms for real-time processing under memory constraints, and data compression techniques for reducing storage and data movement costs.

In this thesis, we contribute to this broader effort by developing more efficient *external memory* and *parallel* algorithms and data structures for fundamental problems. The external memory model captures the behavior of algorithms operating on large datasets that must primarily reside in external storage. In such cases, the time to transfer data between external storage and main memory becomes the dominant cost. The model incorporates the notion of locality of reference [54], which is the observation that it is significantly faster to access data that is stored close together rather than scattered data. To illustrate, the reader might imagine grocery shopping, where picking up all the items from one aisle at a time is much faster than zigzagging through the store for each item individually. Parallel algorithms seek to solve problems more quickly by leveraging multiple computational units that can operate concurrently. Their design introduces additional challenges, including the need for careful synchronization, efficient communication, and management of concurrent operations on shared resources.

Data structures are essential for managing large datasets, as they enable efficient updates and queries. To this end, a central aim of this thesis is to develop *persistent*

data structures, particularly in the external memory model. A standard data structure is *ephemeral*, meaning it only allows access to its present version. In contrast, persistent data structures also allow queries, and possibly updates, to any of their previous versions. In effect, each update derives a new version of the structure. To give an example, for a social media platform maintaining a data structure for its users, an update could be the insertion or deletion of a user account, and a query could be to report all users with between 100 and 1000 connections. If the data structure were persistent, the query could also be qualified with a time, asking for the users with between 100 and 1000 connections on a specific date and time. A trivial way to achieve persistence is to copy the entire structure upon each update and apply the change to the new copy. A classical result by Driscoll et al.[58] shows that a wide range of linked data structures can be made persistent with no asymptotic space or time overhead. In this thesis, we make significant progress on the decades-old problem of creating persistent search trees in the external memory model. A key part of our approach is to embrace a geometric interpretation of persistence and leverage computational geometry data structures.

The final component of this thesis presents contributions to *computational geometry*. First, we present an outlier detection algorithm for two-dimensional data that is based on convex hull peeling. The peeling order is determined by how much each point contributes to the area of the convex hull. The underlying principle is that a point is more likely to be an outlier if its removal from the dataset results in a significant decrease in the area of the convex hull. Outlier detection typically serves two key purposes. In some cases, outliers are discarded or corrected to ensure data quality for subsequent analysis. For example, if a fitness tracker records that a user took $50,000$ steps in one hour, the value might be adjusted based on typical activity patterns. Alternatively, outliers represent meaningful deviations that warrant further analysis due to their anomalous or novel nature. Examples include identifying fraudulent financial transactions [101] or discovering new physics by detecting unusual particle collision events in particle accelerators, which could indicate phenomena beyond the Standard Model [133]. Second, we present a polynomial-time algorithm for the contiguous art gallery problem, where only the boundary must be guarded and each guard is restricted to covering a contiguous portion of it. Whereas the original art gallery problem and many of its variants are computationally hard, our result helps clarify the boundary between tractable and intractable variants of the problem. We demonstrate that a remarkably simple greedy algorithm, which employs only known computational geometry techniques, finds an optimal set of guards in polynomial time.

1.1 An Argument for Theoretical Computer Science

Theoretical computer science as a distinct scientific discipline emerged less than a century ago with the introduction of the Turing machine, which provided a simple yet precise mathematical model of computation. This formalization enabled researchers

to prove fundamental results about the limits of computation, including the existence of undecidable problems [130] and the NP-completeness of Boolean satisfiability as established independently by Cook and Levin [46, 97]. [1] Although highly theoretical, these results remain foundational and continue to influence practical advances. They motivate the development of heuristics and approximation algorithms for intractable problems [20, 70, 72, 135], underpin the security of modern cryptography [86], and enable advances in formal verification and model checking [45, 83, 103]. A central reason for their enduring impact is the use of abstract, mathematically rigorous models for defining computation and for measuring its efficiency. Early pioneers could not have anticipated the decades of dramatic advances in hardware design and system architecture. Nevertheless, many of the insights from these models remain remarkably relevant.

Indeed, the power of applying established theoretical models to modern computational challenges is exemplified by the recent FlashAttention algorithm [50, 51, 121], which accelerates attention algorithms in Transformer models, a central architecture in modern machine learning. The authors observed that on modern GPU hardware, attention mechanisms are often bottlenecked by memory transfer between slow high-bandwidth memory (HBM) and fast on-chip SRAM, rather than by arithmetic computation. By analyzing and optimizing their algorithm within the decades-older external memory model, they achieved an order-of-magnitude speedup in practice, primarily through optimized memory access patterns between HBM and SRAM. The resulting paper has since received more than 2000 citations. This illustrates the robustness and continued relevance of the external-memory model, especially considering that it predates the introduction of modern GPUs in the late 1990s.

1.2 Overview of Manuscripts and Contributions

This thesis seeks to balance theory and practice. Although the included publications are theoretical in nature, their focus on external memory and parallelism reflects a commitment to addressing real-world computational constraints. The results and techniques in these works may serve as a starting point for practitioners. Some manuscripts, such as Manuscripts 10 and 11, lie at the intersection of theory and practice, including concrete implementations and experiments. Part II contains the following 7 publications:

Chapter 5
External-Memory Priority Queues with Optimal Insertions [38]
Gerth Stølting Brodal, Michael T. Goodrich, John Iacono, Jared Lo, Ulrich Meyer, Victor Pagan, Nodari Sitchinava, and Rolf Svenning.
To appear in the Proceedings of the 33rd Annual European Symposium on

[1] These developments also gave rise to the famous P versus NP question, one of the seven Millennium Prize Problems for which the Clay Mathematics Institute offers a reward of $1,000,000$ [42].

Algorithms (ESA 2025).
Covered in Part 1 Section 2.3.

Chapter 6
Buffered Partially-Persistent External-Memory Search Trees [39]
Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning.
To appear in the Proceedings of the 33rd Annual European Symposium on Algorithms (ESA 2025).
Covered in Part 1 Section 2.6.

Chapter 7
External Memory Fully Persistent Search Trees [35]
Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning.
Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC 2023).
Covered in Part 1 Section 2.7.

Chapter 8
Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location [34]
Gerth Stølting Brodal, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, and Rolf Svenning.
Proceedings of the 18th International Symposium on Algorithms and Data Structures (WADS 2023).
Covered in Part 1 Section 2.8.

Chapter 9
Polynomial-Time Algorithms for Contiguous Art Gallery and Related Problems [23]
Ahmad Biniaz, Anil Maheshwari, Magnus Christian Ring Merrild, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, Eliot W. Robson, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, Thomas Shermer, Jack Spalding-Jamieson, Rolf Svenning, and Da Wei Zheng.
Proceedings of the 41st International Symposium on Computational Geometry (SoCG 2025).
Covered in Part 1 Section 4.2.

Chapter 10
Fast Area-Weighted Peeling of Convex Hulls for Outlier Detection [127]
Vinesh Sridhar and Rolf Svenning.
Proceedings of the 36th Canadian Conference on Computational Geometry (CCCG 2024).
Covered in Part 1 Section 4.1.

Chapter 11
The All Nearest Smaller Values Problem Revisited in Practice, Parallel and

External Memory [125]
Nodari Sitchinava and Rolf Svenning.
Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2024).
Covered in Part 1 Section 3.1.

Although the author has played a substantial role in all joint work presented here, we highlight some particularly significant contributions for each chapter as required by the Graduate School guidelines. Chapter 5 describes an external memory priority queue with fast insertions. For this chapter, the author wrote the initial draft of the full publication, including technical proof details. In Chapter 6, which describes how to combine partial persistence with buffers in external memory, the author contributed significantly to Section 3 on achieving worst-case bounds. Chapter 7 contains the full version of the publication on fully persistent search trees in external memory. We show that full persistence can be achieved without asymptotic overhead compared to classical B-trees [16]. The author's contributions included giving the conference presentation and developing the work in Sections 4–6 on external memory point location, colored predecessor queries, and global rebuilding. Chapter 8 describes how to adapt the node copying technique for partial persistence to the functional model where all objects are immutable. The author of this thesis contributed in particular to Section 1 on related work and Lemma 1 which limits cascading due to node copying. Chapter 9 shows that the contiguous art gallery problem belongs to the complexity class P. That is, it can be solved in polynomial time on a Turing machine [130]. The main result was obtained independently by Biniaz et al. [24] and Robson et al. [117]. This chapter details the specific version of the result to which the author of this thesis contributed. This approach is based on repeatedly applying a simple greedy algorithm until optimality. The author of this thesis contributed in particular to Section 1 on related work, Section 3 on the GREEDYINTERVAL algorithm, and the initial outline Sections 4 and 5 on the combinatorial and geometric behavior of the algorithm. Chapter 10 describes an efficient convex hull peeling algorithm that repeatedly removes the point from the convex hull that causes its area to decrease the most. For this chapter, the author contributed in particular by posing the initial research question and developing and analyzing the final algorithm. Chapter 11 improves the analysis of an existing heuristic algorithm for the all nearest smaller values problem, showing that it is work-efficient, similar to a more complex theoretical algorithm [22]. We present the first implementation of the theoretical algorithm, and show that it performs well in practice. For this chapter, the author's contributions included posing the central conjecture that the heuristic algorithm is work-efficient, outlining the corresponding proof strategy, contributing substantially to the writing of the manuscript, performing the experiments, and giving the conference presentation.

Chapter 2

External Memory and Persistence

This chapter provides the necessary background for the first four manuscripts of this thesis. We begin in Section 2.1 by reviewing the classical random access models to establish a theoretical baseline. This is followed in Section 2.2 by a review of the external memory model. Building upon this foundation, Section 2.3 then presents our first contribution, a novel external memory priority queue with optimal insertions. The remainder of the chapter is dedicated to persistence, beginning in Section 2.4 with a general introduction to the topic, followed by our three contributions to this area. Section 2.6 presents our work on buffered, partially persistent search trees. Section 2.7 then describes a fully persistent search tree that achieves performance bounds matching those of classical B-trees [16]. Finally, Section 2.8 details a space-efficient transformation for functional persistence.

2.1 Random Access Models

Designing efficient algorithms and data structures is central to theoretical computer science. To do so rigorously, the problem and its corresponding model of computation must be defined with mathematical precision. The standard model of computation is the random access model (RAM) consisting of a central processing unit (CPU) with random access to an infinite main memory array. The CPU executes an algorithm or program, which is assumed to have a finite description that is independent of the input size. The input is initially placed in N consecutive cells of the main memory, with one element of the input occupying one cell. All standard CPU operations, such as $+, -, \setminus, *, \leq, =, \geq$ and reading and writing to any index of the main memory, are allowed, and all CPU operations are defined to take unit time and fit in one cell of the array. Almost all the algorithms and data structures in this thesis are *comparison-based*, meaning that the only operations the CPU performs on input elements are comparisons, i.e., $\leq, =, \geq$. Furthermore, the intermediate numbers produced are small enough to fit in one cell of the array, such that all CPU operations take unit time. This restriction to comparison-based algorithms is of particular theoretical interest. It enables the proof of fundamental lower bounds, such as the comparison bound

for sorting N elements [89]. However, on a real computer everything is eventually represented by bits. This is captured by the word-RAM model, where each cell of the array is a word w consisting of $w = \Omega(\log N)$ bits. In this setting, sorting n words is possible in linear time for a wide parameter range for w. For example, sorting numbers of size $\mathcal{O}(N^k)$ for any constant k takes $\mathcal{O}(N)$ time using the classical radix sort algorithm. It is a major open problem whether it is possible to sort in linear time for any value of w [17]. The word-RAM model typically allows AC0 operations in constant time, and sometimes also multiplication, which is not an AC0 operation [69].

The RAM models have several limitations that are relevant to this thesis. First, they do not model locality of reference or the memory hierarchy which is a key part of most real computers, and is captured by the external memory model [5]. Second, they do not model parallelism, except possibly word-level parallelism [17]. The parallel random-access machine model, for example, is a model that captures parallelism [66, 73, 120].

2.2 The External Memory Model

In most modern computer architectures, there is a complex memory hierarchy from registers, on-chip caches, main memory, to secondary storage. Smaller but faster-to-access memory units are placed closer to the processor, and larger but slower-to-access memory units are placed farther away. This design enables the processor to store frequently used data in closer, faster memory. Data is moved between levels of the memory hierarchy in blocks of consecutive elements. This block-based transfer relies on the principle of locality of reference, which observes that spatially close elements are more likely to be accessed within a short time frame. Designing algorithms with memory access patterns and the memory hierarchy in mind has proven essential for many practical applications, including graph algorithms [76], file systems [82], attention algorithms for machine learning [50, 51, 121], and modeling terrain water flow [10, 12, 49, 116].

The external memory model was introduced by Aggarwal and Vitter [5] and captures a simplified two-level hierarchy with a fast internal memory of size M, and an infinite external memory. Both internal and external memory are divided into blocks of B consecutive elements. All computation takes place in internal memory. Data is moved between internal and external memory using I/Os that transfers one block of elements. The input of size N initially resides in $\lceil N/B \rceil$ blocks of external memory, and the algorithms in this model are measured by the number of I/Os they perform. The space usage of an algorithm is the number of blocks of external memory that it uses. The minimal assumptions for the parameters M and B are $B \geq 1$ and $M \geq 2B$, and usually also that $N > M$, since otherwise almost any problem can be solved trivially. The cost measure of the external memory model is highlighted by two fundamental computational problems. Scanning an input of size N requires $\mathit{Scan}(N) := \mathcal{O}(N/B)$ I/Os. Sorting N elements requires $\mathit{Sort}(N) := \mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os. The latter is a tight bound for comparison-based algorithms [5].

An important class of algorithms is *cache-oblivious* algorithms [67, 68], which are designed without explicit knowledge of the parameters M and B, but still analyzed in the external memory model. It is assumed that an optimal offline cache replacement policy is used to control which blocks to keep in internal memory and which to evict. This is a minor assumption, since the least-recently used policy can effectively simulate the optimal strategy [67, 126]. A key advantage of cache-oblivious algorithms is that if they are optimal for two levels of memory, they are also optimal for multiple levels of memory [67, 68]. To achieve optimal comparison-based sorting in the cache-oblivious model, the tall-cache assumption $M \geq B^{1+\varepsilon}$ for any constant $\varepsilon > 0$ is necessary [31]. Simple examples of cache-oblivious algorithms include scanning an array and binary merge sort. However, while binary merge sort is cache-oblivious, it is not optimal in the external memory model. In contrast, $\Theta(M/B)$-way merge sort is optimal but inherently cache-aware. The first optimal cache-oblivious sorting algorithm is *funnelsort* [67].

2.3 Heaps With Fast Insertions

In this section, we describe our contributions from manuscript [38] (Chapter 5) where we present an external memory priority queue with fast insertions.

A priority queue is a fundamental data structure for storing a dynamic multi-set of elements from a totally ordered universe U subject to two simple operations:

- INSERT (e): inserts an element $e \in U$ into the priority queue.
- DELETEMIN: deletes and returns a smallest element in the priority queue.

Priority queues are widely applicable,for example for minimum spanning tree [115] and shortest path algorithms [57], and have been implemented in many models of computation [8, 30, 48, 139].

Priority queues in the RAM model

Heapsort [65, 139] is a classical sorting algorithm for sorting n elements by performing n INSERT operations, followed by n DELETEMIN operations to produce a sorted output. Since sorting requires $\Omega(n \log n)$ comparisons, either INSERT or DELETEMIN must use $\Omega(\log |Q|)$ amortized comparisons, where $|Q|$ is the size of the priority queue just before performing the operation.

The most basic implementation of a priority queue is arguably a binary heap. This is a binary tree that satisfies the heap property, meaning that the key stored at each node is less than (symmetrically greater than) the keys stored at its children, depending on whether it is a min-heap or a max-heap. When the tree is complete, it can be stored compactly in an array, where the parent-child relationships can be determined using simple index arithmetic. It supports INSERT and DELETEMIN both in time and in comparisons proportional to the height of the heap which is $\Theta(\log |Q|)$.

However, this is not ideal, as only one of the operations needs to be slow enough to match the lower bound. Since the number of DELETEMIN operations is always at most the number of INSERT operations, it is straightforward to achieve DELETEMIN in amortized $\mathcal{O}(1)$ comparisons and INSERT in $\mathcal{O}(\log n)$ comparisons, by charging the cost of a DELETEMIN to a previous INSERT. It turns out that it is also possible to flip the cost around, and binomial heaps [137] achieve DELETEMIN in amortized $\mathcal{O}(\log n)$ comparisons and INSERT in amortized $\mathcal{O}(1)$ comparisons. This is useful in any application where the priority does not end up empty, i.e. not every inserted element is deleted. Numerous priority queues have been developed, and other standard operations include:

- DECREASE-KEY (e, e'): replaces element e with e' given a pointer to element e provided $e \leq e'$.
- MELD: merges two heaps into one.
- DELETE (e): deletes element e given a pointer to it.
- MAKE-HEAP (S): initializes a heap for the multi-set of elements S.

Fibonacci heaps were the first priority queue, achieving a constant amortized number of comparisons for all operations except DELETE and DELETEMIN. A long line of work culminating in the Brodal Queue [30] and Strict Fibonacci heaps [33, 36] achieved the same optimal guarantees in the worst case rather than amortized, in the RAM model and pointer machine model, respectively.

Priority queues in the EM model

Several priority queues have been developed in the external memory model. Examples include a structure by Arge [6] based on buffer trees [6], a tournament tree approach by Kumar and Schwabe [91], a buffered M/B-ary heap by Fadel, Jakobsen, Katajainen, and Teuhola [63], an array heap [129] approach by Brengel, Crauser, Ferragina, and Meyer [28], and the worst-case efficient array heap by Brodal and Katajainen [32]. They all perform INSERT and DELETEMIN using $\mathcal{O}\left(\frac{1}{B}\log_{M/B}|Q|\right)$ I/Os, all amortized except the structure by Brodal and Katajainen. Similarly to the RAM model, these priority queues can form the basis of an external-memory Heapsort algorithm, and hence INSERT or DELETEMIN must use amortized $\Omega\left(\frac{1}{B}\log_{M/B}|Q|\right)$ I/Os. Since each deletion can be fully charged to a preceding insertion, the amortized I/Os of DELETEMIN can be restated as zero. In manuscript [38], we show that it is possible to swap the efficiency of INSERT and DELETEMIN in the external memory model, similar to what binomial queues achieve in the RAM model, to achieve fast insertions and slow deletions. Our structure is simultaneously optimal with respect to the amortized number of I/Os, comparisons, and time.

Theorem 1 (Manuscript [38] Theorem 1) *There exists an external-memory priority queue supporting* INSERT *with amortized $\mathcal{O}(1)$ comparisons and $\mathcal{O}\left(\frac{1}{B}\right)$ I/Os, and*

DELETEMIN *with amortized* $\mathcal{O}(\lg N)$ *comparisons and* $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ *I/Os, where* N *is the current number of elements in the priority queue. The space usage is* $\mathcal{O}\left(\frac{N}{B}\right)$ *blocks. The memory size only needs to satisfy the minimal requirement* $M \geq 2B$.

We focus on developing a data structure, which is efficient with respect to the number of INSERT operations I and DELETEMIN operations D.

Lemma 2 (Manuscript [38] Lemma 2) *There exists an external-memory priority queue supporting a sequence of* I INSERT *and* D DELETEMIN *operations, using a total of* $\mathcal{O}(I + D\lg I)$ *comparisons and* $\mathcal{O}\left(\frac{1}{B}\left(I + D\log_{M/B}\frac{I}{B}\right)\right)$ *I/Os, assuming* $M \geq 2B$.

Using the standard global rebuilding technique [112] together with Lemma 2, we ensure that the number of insertions in the current structure is close to the number of elements present in it, which leads to Theorem 1. To be more precise, consider a structure with N_0 initial insertions, then after $\lceil N_0/2 \rceil$ INSERT or DELETEMIN operations, we collect all elements present in the structure using a single scan, and insert them into a new structure using $\mathcal{O}(N_0)$ comparisons and $\mathcal{O}(N_0/B)$ I/Os by Lemma 2. Within each such step, the number of insertions and the size of the data structure are within a factor of two.

Our structure shares some similarity with the buffered $\Theta(M/B)$-ary heap by Fadel, Jakobsen, Katajainen, and Teuhola [63]. Their structure does not have fast insertions for two reasons. First, the buffers of size $\Theta(M)$ are always sorted, leading to $\Omega(N \log M)$ comparisons just for the buffers at the leaves. Second, whenever a new leaf is added, elements may have to be moved along the entire leaf-to-root path to restore the heap property. We overcome both of the above challenges, which persist even when only insertions are performed. At a high level, our data structure has an internal and external part, and elements are *transferred* in batches of $\Theta(M)$ elements between the two as needed. The external part is for storage and supports efficient batched retrieval of its overall smallest elements. The internal part fits in memory and allows for comparison-efficient INSERT and DELETEMIN operations. If the internal part cannot return the result of a DELETEMIN operation, a transfer is made from the external part to it. Conversely, if an INSERT operation causes the internal part to become too large, a batch of elements are transferred to the external part. See Figure 2.1.

Internal part

The internal part uses $\mathcal{O}(M)$ space and always fits in internal memory. It consists of *min-buffer* S, a pivot element p, and an insertion buffer L. The min-buffer is implemented as an insertion efficient internal memory priority queue S, such as a binomial queue [137]. We maintain the invariant that all elements in S are smaller than p, which in turn is smaller than any remaining element in either the internal or external part. When an element e is inserted, it is inserted in S or appended to L, according

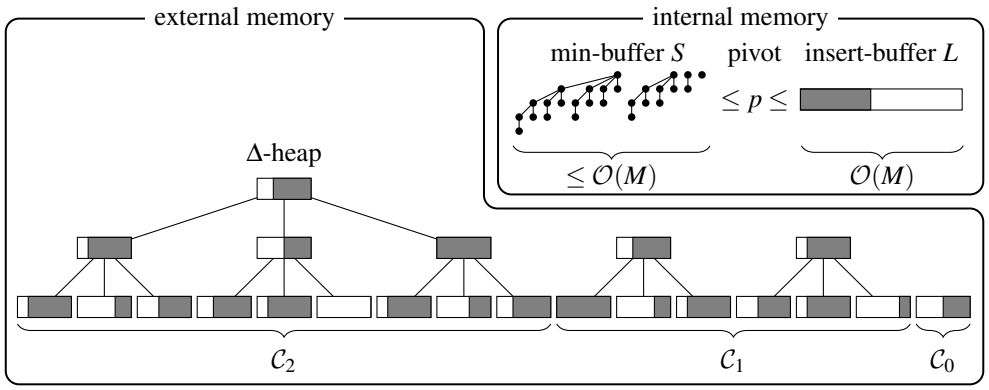


Figure 2.1: (Manuscript 5 Figure 1) The external-memory priority queue. All rectangles are buffers of capacity $\mathcal{O}(M)$, where the gray area illustrates elements in buffers.

to the above invariant. As long as S is nonempty, DELETEMIN simply returns the smallest element in S, incurring no I/Os, and $\mathcal{O}(\log n)$ comparisons. If S is empty, then the $\Theta(M)$ smallest elements in the external part are transferred to the internal part to fill S accordingly. If the internal part is full, $\Theta(M)$ elements are transferred to the external part from either S or L. If the transfer is from S, the larger half of its elements are identified using the linear-time median finding algorithm by Blum et al. [27] and transferred, and S is rebuilt by inserting the smaller half of elements into an empty structure. Further details are omitted.

External part

The key structure in the external memory part is a Δ-*heap* , which is a Δ-ary tree where all leaves are at the same height and all internal nodes have exactly Δ children. In our setting, $\Delta = \Theta(M/B)$, and each node of the tree has a buffer with the capacity to store M elements. A Δ-heap satisfies the *extended heap order*, meaning that for any node, the elements stored in its buffer are smaller than or equal to any element stored in the buffers of its children. It can be seen as a generalization of the classic binary heap [139], augmented with buffers, and is similar to the external-memory priority queue proposed by Fadel et al. [63]. However, a key distinction in our approach is that we do not require buffers to be internally sorted, as that would be incompatible with achieving constant-time insertions. The buffers are stored as *semi-sorted* lists in internal nodes.

Definition 3 (Semi-sorted lists) *A semi-sorted list is a linked list of blocks b_1, b_2, b_3, b_δ, where all blocks contain exactly B elements, except the first and last block, which contain $\leq B$ elements. It is partially sorted, with all elements in b_i being smaller than or equal to all elements in $b_i + 1$, for $1 \leq i < \delta$.*

Even storing the buffers at the leaves as semi-sorted lists would be prohibitive for fast insertions. Instead, at the leaves, buffers are stored as lazy semi-sorted lists,

which are incrementally semi-sorted depending on how many elements have been accessed in the buffer. Figure 2.2 illustrates this structure.

Definition 4 (Lazy semi-sorted lists) *A lazy semi-sorted list of M elements is organized into a sequence of chunks $c_1, c_2, ..., c_\delta$, where $\delta = \lceil \lg \frac{M}{B} \rceil + 1$. All elements in chunk c_i are smaller than or equal to all elements in chunk c_{i+1}, for which we use the notation $c_i \preceq c_{i+1}$, but no order is required internally in a chunk. Chunk c_i stores $B2^{i-2}$ elements for $2 \leq i < \delta$, and chunk c_δ stores the remaining elements. The first time a chunk is accessed, the chunk is semi-sorted.*

Since buffers, and thus chunks, in a Δ-heap are always accessed from left to right (from smaller to larger elements), the cost of semi-sorting the chunks can be amortized over the number of elements accessed within the buffer.

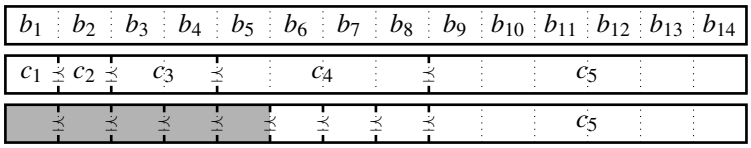


Figure 2.2: A lazy semi-sorted list. **Top:** Initial unsorted list of blocks $b_1, \ldots, b_{14}$. **Middle:** Initial chunks $c_1 \preceq \cdots \preceq c_5$. **Bottom:** The state after the first five blocks $b_1, \ldots, b_5$ in the semi-sorted list have been accessed, i.e., the chunks $c_1, \ldots, c_4$ have been made into semi-sorted sublists.

In a Δ-heap , elements are always pulled toward the root. More precisely, it satisfies the invariant **I1** that each buffer at a node u contains $\Theta(M)$ elements, unless no elements are stored at any descendant of u. The extended heap order, together with **I1**, implies that $\Theta(M)$ smallest elements in a Δ-heap can be found at its root, unless all elements in the Δ-heap are stored at the root. When elements are transferred from the external part to the internal part, invariant **I1** might be violated for a node u (initially a root) of a Δ-heap . The invariant is restored by recursively *pulling* elements from the children of u, by moving some elements from the children of u, to the buffer of u.

Modifying the leaf of a Δ-heap is problematic, since restoring the extended heap order by examining the leaf-to-root path is prohibitive to achieve fast insertions. Instead, we keep collections $C_0, C_1, ...C_h$ of Δ-heaps , where the Δ-heaps in collection C_i have height i, the size of each collection is $|C_i| < \Delta$, and $h = \mathcal{O}\left(\log_{M/B} I/M\right)$. In this way, $\Theta(M)$ elements are added to the external part by insertion them in collection C_0 as a Δ-heap of height 0 (the root is the only node). Similar to a addition in a Δ-ary number system, when a collection C_i reaches a size of exactly Δ, all its Δ-heaps are merged under a new root with an initially empty buffer, and the resulting Δ-heap is added to collection C_{i+1}. The extended heap order and invariant **I1** guarantees that the $\Theta(M)$ smallest elements of the external part can be found by examining the $\mathcal{O}\left(M \log_{M/B} I/M\right)$ elements in the $\mathcal{O}\left(\Delta \log_{M/B} I/M\right)$ buffers at the roots of the Δ-heaps in all collections.

Analysis

In this section, we sketch the key ideas underlying Lemma 2. Whenever INSERT or DELETEMIN can be handled entirely within the internal part and trigger no transfers, the bounds of Lemma 2 follow directly when an insertion-efficient priority queue such as a binomial queue is used. Transfers trigger additional work in both the internal and external part, and the first part of the analysis is to bound the number of transfers as a function of the number of INSERT operations I and DELETEMIN operations D.

Lemma 5 (Manuscript [38] Lemma 7) *The number of transfers from the internal to the external part is $\mathcal{O}(D/M)$. The number of transfers from the external to the internal part is $\mathcal{O}(I/M)$.*

The key idea is that each transfer in either direction moves a batch of $\Theta(M)$ elements to rebalance the internal part. Each INSERT operation gradually increases the internal load, eventually triggering a transfer to external memory once capacity is exceeded. Conversely, DELETEMIN operations deplete the internal priority queue, and a transfer from external memory is only needed when it becomes empty. Hence, each transfer is separated by $\Omega(M)$ operations of the same type.

The remainder of the analysis revolves around bounding the number of pulls to restore invariant **I1** in the external part and the corresponding costs in comparisons and I/Os.

Lemma 6 (Manuscript [38] Lemma 8) *The total number of pulls is $\mathcal{O}\left(\frac{I}{M\Delta}+\frac{DH}{M}\right)$ and their total cost is $\mathcal{O}\left(I\frac{\lg\Delta}{\Delta}+D\lg\frac{I}{M}\right)$ comparisons and $\mathcal{O}\left(\frac{I}{M}+\frac{DH}{B}\right)$ I/Os, excluding the cost of accessing the leaves.*

We bound the number of pulls performed at a given height h by bounding the total size of all buffers at heights $> h$, along with the number of transfers from the external part to the internal part (Lemma 5). This also yields a bound on the number of elements accessed at the leaves, and hence the degree to which the lazy semi-sorted buffers at the leaves have been semi-sorted. We then bound the comparisons and I/Os involved in this process and show that it is possible to charge the nonlinear cost to the DELETEMIN operations.

Lemma 7 (Manuscript [38] Lemma 9) *The total cost of constructing the lazy semi-sorted leaves and accessing them is $\mathcal{O}(I+D\lg\Delta)$ comparisons and $\mathcal{O}\left(\frac{I}{B}+\frac{D}{B}\right)$ I/Os.*

The initial exponentially increasing sized chunks of the lazy semi-sorted list are constructed in order from the largest to the smallest, by applying the linear-time and linear-I/O selection algorithm by Blum et al. [27]. The chunks are semi-sorted as they are accessed by recursively applying the selection algorithm by Blum et al. [27], each time roughly partitioning the input into two balanced parts, until reaching a base case of B elements.

Open Questions

An obvious way to improve the results in this work is to design a structure with worst-case guarantees. A major challenge here is the recursive pulling needed to fill the buffers in the external part. It is possible that classical deamortization techniques based on cup games [55, 56] could prove useful here.

A different direction is to design a cache-oblivious structure. This would require significant changes since our structure is inherently cache-aware with the size of the internal part, buffer sizes in the external part, and the degree of nodes in the external part being defined in terms of M and B.

2.4 Persistent Data Structures

Normally, when a data structure undergoes changes through a sequence of updates, only the present version is stored and available for queries. Such a data structure is known as *ephemeral*. In contrast, when a *persistent* data structure is updated, it simply derives a new version of the data structure where the update has been applied, while still allowing queries in any previous version. There are many notions of persistence, and in this thesis we consider two. The first is *partial persistence*, which allows queries on any version but restricts updates to only the most recent one. In this case, the versions of the data structures form a path. The second is *full persistence*, which permits both queries and updates on any version. We emphasize that when an earlier version is updated, conceptually a copy of that version is made, and then the copy is updated. In this case, the versions of the data structure form a tree as in Figure 2.4.

A simple but general way to achieve persistence is to explicitly copy the data structure and then apply the update to the copy. This approach preserves the query time of the underlying data structure, but is dramatically inefficient in terms of update time and space. In the other extreme, by storing only the sequence of updates and then for a query performing the updates, optimal space and constant-time updates can be achieved, but with dramatically inefficient queries. A time-space trade-off between these two naive approaches is possible by only copying every k'th version, and was described by Overmars [111, 112].

Driscoll et al. devised an efficient general transformation to make any ephemeral linked data structure fully persistent, as long as the indegree of every node is constant [58]. A key idea was to attach timestamps to the edges of the data structure, allowing logical rather than destructive changes to the structure. The transformation uses linear space in the number of changes to the underlying data structure. It also asymptotically preserves the update and query time of the underlying data structure, with respect to the specific version that is accessed. More precisely, consider a persistent data structure that undergoes n updates. A query in version v_i, for $1 \leq i \leq n$, takes time proportional to the size of version v_i, rather than n (or i), assuming that the underlying query complexity depends on the size of the data structure.

2.5 Persistent Search Trees in External Memory

In this section, we present the results from Manuscripts 6 and 7 on persistent search trees in the external memory model. Before these works, the state-of-the-art persistent search trees in external memory were based on adapting the general transformations for persistence by Driscoll et al. Central to both of our results is a two-dimensional, geometric interpretation of persistence. This approach stands in contrast to previous graph-based methods that directly manipulate the underlying linked data structure.

Search Trees

A fundamental data structure problem is to store a totally ordered set S subject to some or all the following operations:

- Updates
  - INSERT(x) Adds x to S, i.e., sets $S = S \cup \{x\}$.
  - DELETE(x) Removes x from S, i.e., sets $S = S \setminus \{x\}$.
- Queries
  - SEARCH(x) Returns the predecessor of x in S, i.e., the largest element in S smaller than x.
  - RANGE(x,y) Reports all elements in $S \cap [x,y]$ in increasing order, i.e. all elements in S that are in the interval $[x,y]$.

In internal memory, a standard approach is to store S using a balanced binary search tree such as AVL [4] trees, red-black trees [74], (a,b) trees [80], and many others. They support all operations in $\mathcal{O}(\log N + K)$ time, where $N = |S|$, and K denotes the size of the output, which is only relevant for range queries. There are additional operations that could be relevant, such as membership testing, rank queries, and finger queries.

In external memory, the classical solution is the B-tree by Bayer and McCreight [16] from 1970. It has nodes of degree $\Theta(B)$, and rebalancing is performed by merging and splitting nodes. It supports all the above operations in $\mathcal{O}(\log_B N + K)$ I/Os. Decades later, Brodal and Fagerberg significantly improved the efficiency of updates by introducing buffers to the nodes of the B-tree, inspired by the Buffer Tree by Arge [6]. Their structure with nodes of degree $\Theta(B^\varepsilon)$ is known as a B^ε-tree, and performs queries in amortized $\mathcal{O}\left(\frac{1}{\varepsilon}\log_B N\right)$ I/Os and updates in amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N\right)$ I/Os. The amortization is due to the flushing of buffers. A line of work has demonstrated that the amortized bounds can be improved. Bender et al. [19] showed that the bounds can be achieved with high probability using randomization. Subsequently, Das et al. [52] showed that similar bounds can be obtained in the worst case. Both results rely on a stronger assumption on the internal memory size M than the standard lower bound of $M \geq 2B$. For typical parameters such as $\varepsilon = 1/2$ and $B = 1000$, the factor

$\frac{1}{\varepsilon B^{1-\varepsilon}}$ is significant, and the idea of lazily updating a structure by buffering updates has found significant applications in industry-strength software such as TokuDB [18] and BetrFS [82].

2.6 Partial Persistence with Buffers

We now outline the contributions of manuscript 6 on improving partially persistent search trees in external memory. Our motivation for this work is straightforward. Prior to our results, the state-of-the-art persistent search trees in external memory were developed by Becker et al., Varman et al., and Arge et al. These structures support queries and updates in worst-case $\mathcal{O}(\log_B N_v + K/B)$ I/Os, where N_v denotes the size of the version being operated upon, and K is the size of the output, relevant only for range queries. All previous approaches essentially adapt the general transformation by Driscoll et al. [58] to B-trees without asymptotic overhead.

In our work, we show that it is possible to design a persistent search tree whose performance matches the optimal bounds of the B^ε-tree.

Theorem 8 (Manuscript 6 Theorem 1) *Given any parameter $0 < \varepsilon < 1$ and $M \geq 2B$, there exist partially-persistent external-memory search trees over any totally ordered set, that support* INSERT *and* DELETE *in amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v\right)$ I/Os,* SEARCH *in amortized $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v\right)$ I/Os, and* RANGE *in amortized $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v + K/B\right)$ I/Os. The space usage is linear in the total number of updates.*

In Section 2.6, we also show how to deamortize the structure, i.e., achieve worst-case bounds. We consider both the mild assumption on the size of the internal memory that $M = \Omega\left(B^{1-\varepsilon} \log\left(\max_v N_v\right)\right)$ and the minimal assumption that $M \geq 2B$. Our result is the first successful combination of persistent and buffered data structures, and we outline our construction in the following sections.

The Geometric View of Partial Persistence

We fully embrace a natural geometric view of partial persistence, in which the state of the data structure across all its versions is represented in two dimensions. This geometric perspective was first explored by Kolovson and Stonebraker [90]. Specifically, we encode updates as segments in the plane, where the horizontal axis represents values and the vertical axis represents versions. Under this interpretation, the ith update operation INSERT(x) corresponds to adding the vertical segment $((x,i),(x,\infty))$. Conversely, the ith update operation DELETE(x) modifies the existing infinite segment intersecting (x,i) to end at (x,i). Thus, in partially persistent structures, new segments always emerge from the top of the plane. The queries SEARCH and RANGE are then naturally interpreted as geometric operations. Specifically, a SEARCH query corresponds to horizontal ray shooting, while a RANGE query corresponds to reporting intersections between a horizontal query segment and vertical segments. An illustrative example is shown in Figure 2.3 (left).

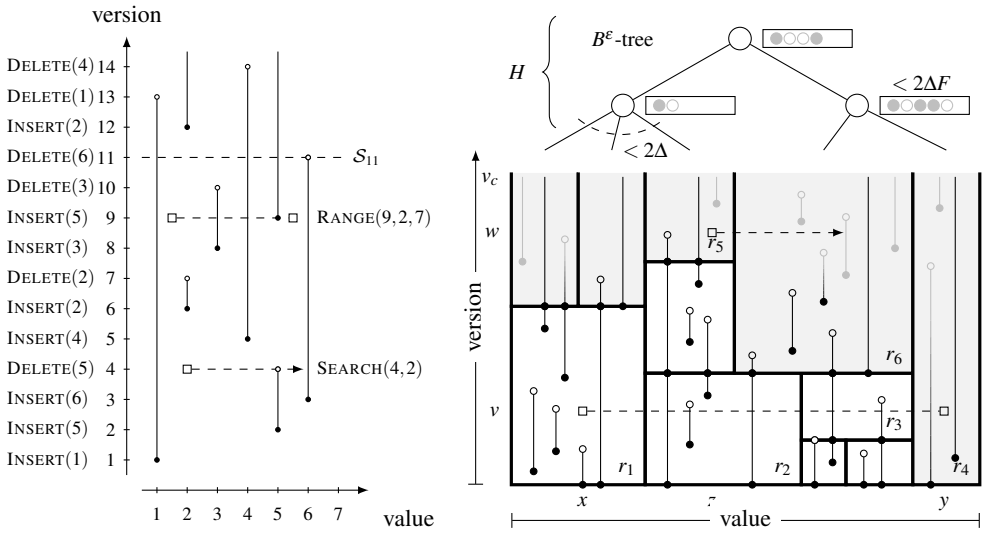


Figure 2.3: (Manuscript 6) **Left**: A list of updates performed on an initially empty set and the geometric interpretation of the updates. Vertical lines illustrate the half-open intervals of versions containing a key. Note that the key 3 is contained in versions $[8,10[$, i.e., versions 8 and 9, whereas the key 2 is contained in version 6 and versions $[12,\infty[$. The topmost dashed line shows that version 11 of the set is $\mathcal{S}_{11} = \{1,4,5\}$, the dashed line segment at version 9 shows that the result of the query $\textsc{Range}(9,2,7)$ is $\{3,4,5\}$, and the bottommost dashed arrow shows that the result of the successor search $\textsc{Search}(4,2)$ is 6. **Right**: A B^ε-tree of the open rectangles and below the geometric interpretation of the updates, split into multiple rectangles, where gray rectangles are open rectangles. The black endpoints represent updates present in the rectangles and the gray endpoints represent buffered updates present in the buffers at the internal nodes of the B^ε-tree. A query $\textsc{Range}(v,x,y)$ is represented as the dashed line between two square endpoints, spanning rectangles r_1, r_2, r_3, and r_4, and a successor query $\textsc{Search}(w,z)$ is represented as the dashed arrow from a square endpoint, spanning two rectangles r_5 and r_6. Black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

Outline of Data Structure

We implement this geometric view explicitly, inspired by the construction for full persistence in manuscript 7. The plane is partitioned into rectangles, each storing $R = \Theta\big(\frac{1}{\varepsilon} B \log_B \overline{N}\big)$ segments. Here, $\overline{N}$ is a parameter, and all versions have size $\Theta\big(\overline{N}\big)$. We ensure this using global rebuilding [112, 113], which creates multiple copies of the structure, each with a different value for $\overline{N}$. Details of this procedure are omitted here, and we describe the construction for a single copy. Rectangles at the top of the plane are called *open*, and all rectangles below them we call *closed*. Open rectangles are stored as the leaves of a B^ε-tree T, allowing updates to trickle

down towards them using buffers. Segments within an open rectangle r may not completely reflect the geometric view yet, as some updates to r remain buffered in T. On the other hand, all closed rectangles have received all their relevant updates, and match the geometric view exactly. When an open rectangle is merged or split, it is *actualized*, and new open rectangles are started above it. When a rectangle is actualized, all updates going towards it on the root-to-leaf path in T, are applied to the rectangle. Queries that intersect a closed rectangles simply inspects the segments in the rectangle directly. Queries intersecting open rectangles first actualize them to account for buffered updates in T. See Figure 2.3 (right) for an example. To locate a rectangle given a query point, we use a point location data structure P, implemented as a partially persistent B-tree on the bottom of the current set of rectangles. Since there are $\mathcal{O}(\overline{N}/R)$ rectangles, we can afford to simply use path copying [58] for persistence. For a SEARCH query, we need to find at most 2 rectangles, the rectangle intersecting the query point, and potentially its neighbor. For a RANGE query, we find each rectangle intersecting the horizontal query segment using a point location query to P. To ensure the $\mathcal{O}(\log_B \overline{N})$ I/Os spent finding each rectangle can be charged to the output, we enforce an additional invariant that a constant fraction of segments within each rectangle must be *spanning*. A spanning segment goes from the bottom to the top of the rectangle, i.e., intersects all versions relevant to the rectangle. Therefore, all but the leftmost and rightmost rectangles contribute $\Omega(R)$ values to the query output, justifying the charging to the K/B term. Since new open rectangles initially contain only spanning segments, we can maintain the invariant by simply closing the rectangle after $\mathcal{O}(R)$ updates. Furthermore, by not closing it before $\Omega(R)$ updates, we get a linear space bound. When closing a rectangle $\mathcal{O}(1)$ merges and splits suffice, and the precise constants for the rectangles are stated in manuscript 6.

Deamortization

The primary challenge in previous work on deamortizing B^ε-trees was to handle *cascading flushes*. That is, when internal nodes of a B^ε-tree are merged, then so are the buffers, resulting in a large buffer which requires many flushes potentially in many directions. These flushes may trigger more merges, leading to further flushes. We circumvent this problem by never merging internal nodes of T. Instead, we bound the height of T using global rebuilding. Furthermore, previous work required that buffers along a root-to-leaf path in T fit entirely in internal memory, thus assuming $M = \Omega(\frac{1}{\varepsilon} B \log_B \overline{N})$. Under this assumption, actualizing a rectangle can be performed in linear I/Os by sorting internally all updates along a root-to-leaf path. Since the root has $\Theta(B^\varepsilon)$ children, at least $\Omega(B^{1-\varepsilon})$ updates must be directed towards some child. Consequently, by flushing towards the child receiving the most updates, at least $\Omega(B^(1-\varepsilon))$ updates must occur between each flush at the root. This ensures that the work triggered by updates can be performed incrementally.

We further show that by viewing the buffer at an internal node as a cup-game [55, 56], the number of updates directed towards any child is bounded by $\mathcal{O}(B^{1-\varepsilon} \log_2 B^\varepsilon)$. This improves the assumption on the size of the internal memory to $M = \Omega(B^{1-\varepsilon} \log_2 \overline{N})$,

leading to the following theorem.

Theorem 9 (Manuscript 7) *Given any parameter $0 < \varepsilon < 1$ and if the size of the internal memory satisfies $M = \Omega\big(B^{1-\varepsilon}\log_2(\max_v N_v)\big)$, there exist partially-persistent external-memory search trees over any totally ordered set, that support* INSERT *and* DELETE *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N_v\big)$ I/Os,* SEARCH *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v\big)$ I/Os, and* RANGE *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v + K/B\big)$ I/Os. The space usage is linear in the total number of updates.*

Next, we consider the minimal assumption that $M = 2B$, in which case sorting the buffers on a root-to-leaf path incurs $\gamma = \mathit{Sort}\big(B^{1-\varepsilon}\log_2 \overline{N}\big)$ I/Os. Our work is the first to consider this minimal assumption for worst-case partial persistence.

To avoid a multiplicative overhead on the K/B term of γ for RANGE queries, we propagate all relevant updates down layer-by-layer in T. To limit the number of these updates to roughly $\mathcal{O}((K/R)B) = \mathcal{O}\big(K/\big(\frac{1}{\varepsilon}\log_B \overline{N}\big)\big)$, we ensure that buffers of internal nodes of T with degree 1 to be empty. To achieve this, we modify the buffer size of internal nodes to scale with their degree. However, this reintroduces the possibility of flushing cascades. We avoid this by employing a cup-game on the leaves of T, rebalancing (if necessary) the leaf that has received the most updates, leading to the following theorem.

Theorem 10 (Manuscript 6) *Given any parameter $0 < \varepsilon < 1$ and $M \geq 2B$, there exist partially-persistent external-memory search trees over any totally ordered set, that support* INSERT *and* DELETE *in worst-case $\mathcal{O}\big(\frac{1}{B^{1-\varepsilon}}\big(\frac{1}{\varepsilon}\log_B N_v + \gamma\big)\big)$ I/Os,* SEARCH *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v + \gamma\big)$ I/Os, and* RANGE *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v + \gamma + K/B\big)$ I/Os, where $\gamma = \mathit{Sort}\big(B^{1-\varepsilon}\log_2 N_v\big)$. The space usage is linear in the total number of updates.*

Open Questions

An intriguing open problem is understanding how the size of internal memory impacts worst-case performance. Specifically, it remains open whether it is possible to construct, or alternatively prove impossible, a structure achieving worst-case I/O bounds matching those of B^ε-trees under the minimal assumption that $M = 2B$.

To date, the literature has only considered individual cup-games. However, our structure incorporates a separate cup-game at each node of T. A promising direction for future research would be to investigate dependent cup-games, modeled as directed graphs, where the contents overflow and propagate along directed edges.

2.7 Fully Persistent B-trees

In this section, we present the contributions of manuscript 7 on fully persistent search trees in external memory. While this work predates that of manuscript 6, on a technical level it is more involved, and introduced the key idea of implementing the

geometric perspective of persistence. The main result is Theorem 11, showing that fully persistent B-trees can be achieved without asymptotic overhead compared to classical ephemeral B-trees, resolving an open problem by Vitter [136].

Theorem 11 (manuscript 7) *There exist external-memory fully-persistent search trees supporting* INSERT *and* DELETE *in amortized* $\mathcal{O}(\log_B N_v)$ *I/Os,* SEARCH *in worst-case* $\mathcal{O}(\log_B N_v)$ *I/Os, and* RANGE *in worst-case* $\mathcal{O}(\log_B N_v + K/B)$ *I/Os. The space usage is* $\mathcal{O}(N/B)$ *blocks.*

Our result improves on the previous state-of-the-art by Brodal, Sioutas, Tsakalidis, and Tsichlas [37] by eliminating an additive $\mathcal{O}(\log B)$ term from the update complexity. The first result on fully persistent B-trees in external memory was published by Lanka and Mays [92] in 1991. Previous work adapts the general techniques for making pointer-based data structures persistent by Driscoll et al., whereas we directly implement the geometric view of persistence.

The geometric view of full persistence

For full persistence, earlier versions can also be updated, leading to a *version tree* T rather than a linear path of versions as for partial persistence. An update in an earlier version v does not propagate to its ancestors (this would be a retroactive update [53]). Instead, v is conceptually cloned to v', and v' becomes the left child of v in T. Performing a left-to-right preorder traversal of T derives a *version list* L, which will be one dimension of the geometric view. Crucially, unlike for partial persistence, the version identifiers will be unordered, making it a significant challenge to navigate the geometric view. In fact, the additive overhead of $\mathcal{O}(\log B)$ for the previous state-of-the-art was a result of using the order maintenance structure by Dietz and Sleator [55] to perform a binary search among $\Theta(B)$ versions in L. We explicitly maintain both the version list and the version tree. The latter is needed for global rebuilding, where a subtree containing a large fraction of all updates is cut out. The other dimension is based on value. Under this interpretation, the ith update operation INSERT(x) adds the unit vertical segment from (x,i) to $(x,succ(i))$, where $succ(i)$ is the successor of version i in the version list. Conversely, the ith update operation DELETE(x) removes the unit segment from (x,i) to $(x,succ(i))$. Whereas for partial persistence only the top part of the plane is modified, unit segments can be added or removed anywhere in the geometric view for full persistence. Figure 2.4 shows an example of a version tree and the corresponding geometric view.

As with partial persistence, to efficiently maintain the geometric view, we partition it into rectangles. The rectangles must satisfy a number of invariants, including storing $R = \Theta\big(B \log_B \overline{N}\big)$ segments and having a constant fraction of spanning segments for RANGE queries. To restore invariants, the rectangles are split vertically or horizontally and merged with the neighboring rectangles. Figure 2.5 highlights some of the basic ways to rebalance the rectangles. This introduced the need for further invariants, such as having a constant number of neighbors. A significant challenge in this work was to

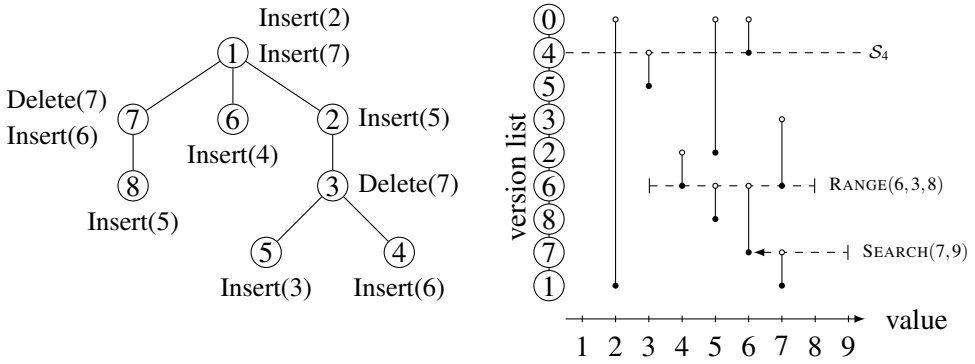


Figure 2.4: (Manuscript 7) **Left:** A version tree illustrating 8 versions of a set after $N = 10$ updates. Versions 1 through 4 contain the sets $\{2,7\}$, $\{2,5,7\}$, $\{2,5\}$, $\{2,5,6\}$, respectively. Updates are shown next to the corresponding node in the tree. **Right:** The geometric view. The version list is obtained by a left-to-right preorder traversal of the version tree terminated by 0. The vertical segments represent half-open intervals indicating the versions in which each value exists. For example, the topmost dashed line shows that version 4 contains $\mathcal{S}_4 = \{2,5,6\}$.

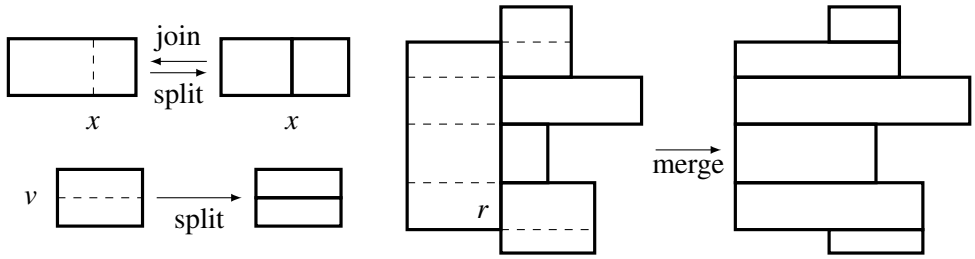


Figure 2.5: (Manuscript 7) **Left:** The primitive rectangle transformations: a vertical split and join at value x, and a horizontal split at version v. **Right:** Merging of a rectangle r with its right neighbors. Dashed lines indicate the horizontal split positions prior to the vertical joins.

show that it was possible to maintain the invariants efficiently in the amortized sense, i.e., to bound the number of times rectangles are merged or split by $\mathcal{O}(N/R)$. This analysis involves 4 potential functions with cyclic dependencies and 12 rebalancing cases. The weighting of each potential is determined using a linear program.

Point Location

To navigate the rectangular partition, we construct a two-dimensional dynamic orthogonal point location structure on the bottom segment of all rectangles with the bounds stated in Theorem 12.

Theorem 12 (Manuscript 7) *There exists an insertion-only point location data structure for non-overlapping horizontal segments, supporting queries in worst-case*

$\mathcal{O}(\log_B N)$ I/Os and insertions in amortized $\mathcal{O}\left(B\log_B^2 N\right)$ I/Os, where N is the number of segments inserted. The space usage is $\mathcal{O}(N\log_B N)$ blocks. Segments are only compared on the vertical axis for equality, that is, testing if they are at the same height.

A significant challenge and obstacle to simply using existing point location structures is that the order on the vertical axis is defined entirely by the version list. We deal with this by devising a structure that only ever compares segments for equality on the vertical axis. Furthermore, our structure must support queries in $\mathcal{O}\left(\log_B \overline{N}\right)$ I/Os, while updates can be significantly slower, since they are only triggered by the merge or split of a rectangle, and the total number of these events is only $\mathcal{O}(N/R)$. We also exploit the observation that it suffices to support insertions, since the area covered by horizontal segments only increases. Table 2.1 shows the various update-query trade-offs, and we note that our structure is the first to achieve queries as fast as traditional *B*-trees. Our structure combines segment trees [21] with a specialized version fractional cascading [44] as a secondary structure at every node to compare versions only for equality. As these secondary structures have to be rebuilt when the nodes split due to insertions, we use a weight-balanced B-tree [7] as the skeleton for the segment tree. Creating an external memory point location structure with $\mathcal{O}(\log_B N)$ I/Os for updates and queries is a challenging and intriguing open problem.

	Update	Query
Arge, Brodal, Rao† [11]	$\mathcal{O}(\log_B N)$	$\mathcal{O}\left(\log_B^2 N\right)$
Munro, Nekrich [105]	$\mathcal{O}(\log_B N \log\log_B N)$	$\mathcal{O}(\log_B N \log\log_B N)$
Brodal, Rysgaard, Svenning‡ [35]	$\mathcal{O}\left(B\log_B^2 N\right)$	$\mathcal{O}(\log_B N)$

Table 2.1: The number of I/Os to perform updates and queries for the different structures, where N denotes the size of the data structure when the query is performed. All structures support amortized updates, and guarantee worst-case query bounds. † Supports general planar subdivisions; the others apply only to orthogonal subdivisions. ‡ Only supports insertions and uses $\mathcal{O}(N\log_B N)$ blocks of space; the others also support deletions and use $\mathcal{O}(N/B)$ blocks of space.

Colored Predecessor

Each rectangle r contains $\mathcal{O}(R)$ vertical segments and thus $\mathcal{O}(R)$ versions from the global version list L. We maintain the versions represented in r in a local version list L^r in the same order that the versions appear in L. This helps us to perform all operations involving r efficiently once it has been found, by comparing versions by their position in L^r. However, for an operation involving version v and rectangle r, then v might not be represented in L^r. Instead of using v we can equivalently use the predecessor of v in L^r, but finding the predecessor is nontrivial since versions

cannot be directly compared as their order is defined by their appearance in L and not by their version identifier. Using the order-maintenance structure by Dietz and Sleator [55] to circumvent this problem reintroduced an additive overhead of $\Omega(\log B)$ that we are attempting to eliminate. Instead, we devised a specialized structure for this problem, which we call the *Colored Predecessor Structure*. We identify each rectangle by a unique integer which we think of as a color. At the leaves of a B-tree we then store the version list L. Furthermore, for each rectangle r containing a version v, we store the *colored version* as a pair (v, r) between v and $succ(v)$. If we conceptually color all leaf-to-root paths from colored versions, then it is straightforward to find the predecessor of any version among the colored versions of a specific color. Note that nodes of the tree will be multi-colored. Figure 2.6a shows an example of this, and Theorem 13 states the precise bounds of the Colored Predecessor Structure.

Theorem 13 (Manuscript 7) *Let N denote the total number of updates to the global version list L and all local version lists L^r. Then there exists a data structure that, given a version $v \in L$ and a color r, can find the predecessor u of v in L^r in worst-case $\mathcal{O}(\log_B N)$ I/Os. The structure supports insertions of versions in both L and L^r and deletions from L^r in amortized $\mathcal{O}(\log_B N)$ I/Os. The space usage is linear in the total number of updates.*

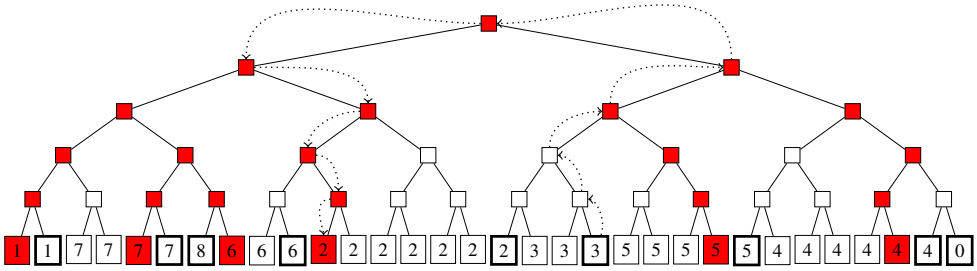
Similarly, but simpler, than the point location structure, we make the structure dynamically efficient using a weight-balanced B-tree [7] and use downpointers [134] to avoid repeatedly searching for the query-color for nodes of the tree. To achieve linear space we do not explicitly color the entire root-to-leaf paths, only the nodes where different paths branch off each other. Figure 2.6b shows an example of this.

Open questions

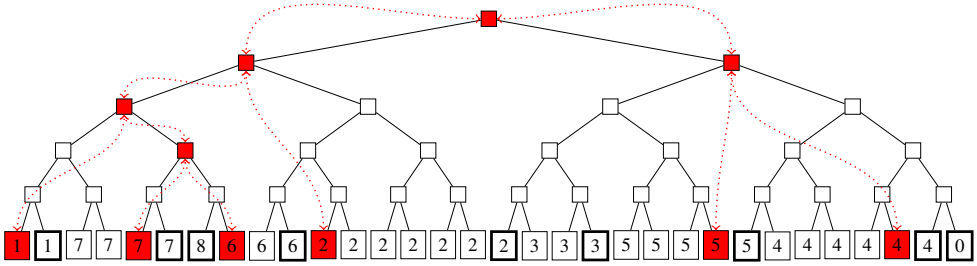
There are two naturally interesting directions for future work. The first is to deamortize the structure and achieve worst-case performance. Both the point location and the colored predecessor structures are amortized. The main obstacle is the rectangle rebalancing, which can cascade dramatically, and as a result, the analysis is inherently amortized. An approach to handle this could be to employ the cup-game technique for deamortization, as we did for buffered partial persistence, by always rebalancing the most unbalanced rectangle. The second direction is to achieve bounds matching the B^ε-tree with significantly faster updates. While this seems more approachable, it remains a significant challenge, as a direct translation of our structure involves buffering both the point location and the colored predecessor structure.

2.8 More Efficient Functional Persistence

In Manuscript 8, we showed how to adapt the general transformation by Driscoll et al. [58] for partial persistence to the *purely functional* model. The seminal work on purely functional data structures is by Okasaki [109]. A central property of the purely



(a) The simple structure where each leaf-to-root path from the red versions are colored. The dotted path indicates the search for the predecessor of version 3 among the red versions.



(b) The efficient structure with only the branches colored.

Figure 2.6: (Manuscript 7) Our simple (2.6a) and efficient (2.6b) colored predecessor structure which for clarity is a binary tree instead of a B-tree and with only one color shown. At the leaves of the tree we have the global version list $1,7,8,6,2,3,5,4,0$ indicated by the leaves in bold. Preceding each version from the global version list we have its copies in arbitrary order.

functional model is that all objects are immutable. Furthermore, data structures are represented as linked structures, that is, directed graphs with a constant number of entry points, implementable in the pointer machine model [58, 128]. Immutability means that the graph cannot be modified, only extended. That is, the graph is changed only by adding new vertices with edges pointing to existing nodes. This constraint naturally results in the graph being acyclic [109, 131]. Tying-the-knot via lazy evaluation can introduce cycles, but we disallow this by assuming a strict functional model [108, 109]. The immutability enforced by the purely functional model has several advantages, including thread safety, easier formal verification, absence of side effects, and inherent persistence of all data structures [15, 81, 109]. Space usage is defined as the number of vertices and edges reachable from the currently active entry points, as if garbage collection were performed automatically and immediately. Vertices can be added or removed as entry points. The efficiency of our general transformation to functional persistence is stated in Theorem 14.

Theorem 14 *(Manuscript 8 Theorem 1 Informal) Any ephemeral, tree-based data structure that satisfies a set of mild structural conditions can be transformed into*

an equivalent purely functional structure supporting offline partial persistence. The transformation preserves the asymptotic update and query times of the original structure. It incurs only an additional $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space overhead for a sequence of n updates. For example, binary search trees, treaps, red-black trees, and functional random access arrays can be adapted to satisfy these conditions with only minor modifications.

Our construction is offline, which means that all updates are applied before any queries are issued. Our transformation enables a linear space, purely functional implementation of the classic slab-based algorithm for planar point location by Sarnak and Tarjan [118].

Ephemeral Transformation for Partial Persistence

Driscoll et al. introduced the *node copying* technique to make ephemeral linked data structures partially persistent. In this technique, each edge is augmented with a *liveness interval*, defined by a start and end timestamp, to support logical rather than destructive updates. For example, if the ith update inserts an edge, it receives the liveness interval (i, ∞). If the jth update, with $j > i$, deletes the edge, the interval is updated to (i, j). Figure 2.7 illustrates this approach. Repeated insertions of edges at a particular vertex may cause its degree to grow arbitrarily, as in the *fat node* technique [58]. This results in a logarithmic overhead for queries, because a query at version i must identify the outgoing edges that are *live* at time i. These are the edges whose liveness intervals include version i. The node copying technique avoids this overhead by splitting nodes so that each node stores at most e outgoing pointers that are not live, where e is a constant. When the threshold is reached, a new node u' is created as a copy of the old node u, but it only includes the edges that are live. Any ingoing edge to u that is live must be duplicated to also point to u'. If the current version is v, then the end timestamp of the edge to u and the start timestamp of the edge to u' are both set to v. Since the edge is duplicated, node copying may cascade. However, an amortized analysis shows that if the maximum indegree of the underlying structure is bounded by a constant $p \leq e$, then the total space and time overhead is linear in the number of changes to the underlying ephemeral structure. The modifications, via timestamps and insertions of edges into existing nodes, are the central challenge in adapting the technique to the purely functional model.

Making the Transformation Functional

Adapting the transformation to the functional model introduces natural restrictions on the underlying ephemeral data structure. Maintaining a directed acyclic graph (DAG) in the functional model is inefficient because logically updating a node requires copying every node that can reach it. Therefore, we restrict the ephemeral data structure to a tree and explicitly maintain only its most recent version. Persistent updates are performed by logically modifying the structure using liveness intervals. Logically deleted edges are not explicitly represented in the current structure. Instead,

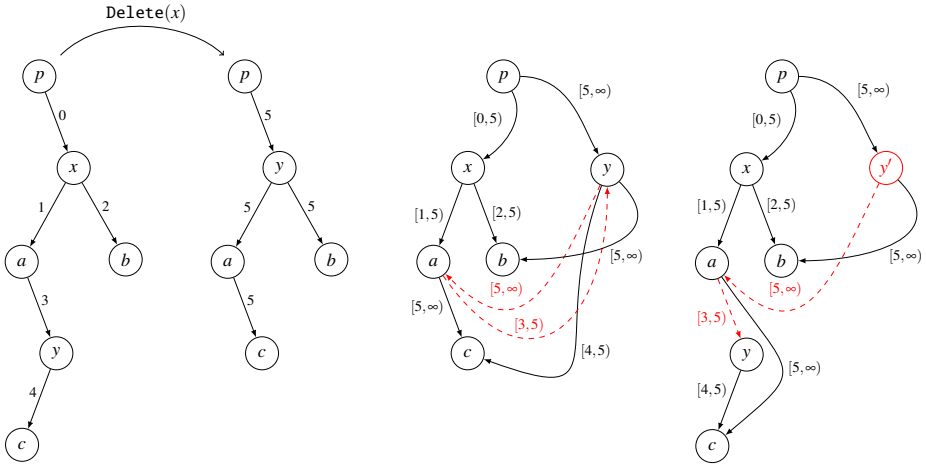


Figure 2.7: (Manuscript 8 Figure 1) A binary search tree (BST) where the value x is deleted and replaced by its successor, without rebalancing. **Left:** An ephemeral BST where a single number above each edge denotes its creation time, shown before and after deleting x. **Middle:** A persistent BST where x is logically deleted from version 5 onward. Each pair of numbers next to an edge denotes its liveness interval. A persistent query for a value includes a version i and follows only the edges that are live at time i. This can introduce a cycle in the persistent graph. **Right:** A modified version of the middle figure, where a new copy y' of y is created so that the resulting graph remains acyclic, as required in the purely functional setting. Cycles are avoided by copying a node that is logically moved upward in the structure during an update.

such edges are stored in a separate list called the *freezer*. As a result, only offline partial persistence is supported, where all updates precede all queries. Since we eventually reconstruct the graph that the ephemeral fat node technique would have produced, we require that the edges stored in the freezer form a DAG. This is necessary because the purely functional setting cannot produce cycles. In some cases, this requires slightly altering how updates are performed in the underlying ephemeral data structure. See Figure 2.7. For technical reasons, we also require that the maximum outdegree of nodes in the ephemeral data structure, as well as the number of edges and nodes created by an update, is constant. These parameters could be generalized, but we omit this for simplicity. After all updates have been performed, all edges from the current structure are added to the freezer. Using the edges from the freezer, we reconstruct essentially the persistent DAG that the node copying technique would have produced online, by interleaving node copying to limit the outdegree introduced by the fat node technique with the topological sorting by Kahn [84]. When a node is processed during the leaves-first traversal of the implicit DAG in the freezer, it points only to nodes that have already been constructed, satisfying the immutability requirement. When a high-degree node is processed and split into multiple lower-degree nodes, similar to node copying, some edges may be duplicated. As in the ephemeral transformation, an amortized analysis shows that the resulting space and time overhead is constant per

update.

Discussion

Although the requirements on the underlying ephemeral data structure may appear restrictive, many standard data structures can be adapted to satisfy them. In particular, the restriction to trees allows persistence to be implemented in a simple way using path copying. The main contribution of our result is that, by adapting the node copying technique to the purely functional setting, we achieve linear space with constant overhead per update. This significantly improves over path copying, which incurs space proportional to the total query time. Our transformation supports offline persistence, and adapting it to support online updates is an interesting direction for future work. One possible approach is to use *implicit edges* to avoid explicitly constructing cycles, which motivated the restriction to the offline setting. An implicit edge is a pair of identifiers that is resolved using a search tree mapping identifiers to nodes. With an implicit edge, there is no direct access between its endpoints, which leads to a logarithmic overhead when navigating the structure. This approach is also amenable to adapting the transformation for full persistence to the purely functional model.

Chapter 3

Parallel Algorithms

Parallel algorithms broadly refer to algorithms in computational models where multiple operations can be performed concurrently. These models include PRAM (Parallel Random Access Machine) [66, 140], BSP (Bulk-Synchronous Parallel) [132], PEM (parallel external memory) [9], circuit models [85], vector models (ultra-wide word-RAM and vector RAM) [25, 64], among others. Key considerations for parallel algorithms include interprocessor communication, synchronization, access patterns to shared resources, and network topology. The diversity of models arises from the desire to capture relevant real-world behavior while keeping the model simple enough for rigorous theoretical analysis. As such, each model emphasizes particular aspects of parallelism while abstracting away others. In this work, we focus on the PRAM model, which assumes synchronous processors accessing a shared memory. It avoids low-level synchronization issues and provides a clean abstraction for theoretical analysis. The PRAM model is similar to the RAM model, but with P processors operating in parallel. Specifically, we use the CREW PRAM variant, which allows concurrent reads (CR) but only exclusive writes (EW). We analyze algorithms in terms of their *work*, which is the total number of operations performed across all processors, and their *span*, which is the time required with an unbounded number of processors. Letting T_P denote the runtime on P processors, the work corresponds to T_1, and the span corresponds to T_∞. By Brent's scheduling principle [29], the runtime on P processors satisfies $T_P = \mathcal{O}(T_1/P + T_\infty)$. A parallel algorithm is called *work-efficient* if its total work matches that of the best known sequential algorithm. Work-efficiency is crucial in practice, as real-world values of P are often limited, and the term T_1/P often dominates the overall runtime.

3.1 The All Nearest Smaller Values Problem

In this section, we describe the contributions of manuscript 11, which investigates the performance of algorithms for the All Nearest Smaller Values (ANSV) problem from both theoretical and practical perspectives. In the ANSV problem, we are given an array of n totally ordered elements, and the goal is to compute, for each

element, the nearest smaller value both to its left and to its right. Each of these is referred to as a *match*. For clarity of exposition, we focus only on right matches in the remainder of this discussion. It is a fundamental parallel problem as it appears as a subroutine in a wide range of applications, including Cartesian tree construction, range minimum queries, and merging sorted sequences. Sequentially, it can be solved in linear time by scanning from left to right while maintaining a stack of unmatched elements. However, as with prefix sums, this approach is inherently sequential, and a fundamentally different strategy is required to achieve parallelism [78]. Berkman, Schieber, and Vishkin [22] provided a work-efficient algorithm with $\mathcal{O}(\log n)$ span in the CREW PRAM model. The algorithm uses the idea of constructing a *min-binary tree* on the input, where each node stores the minimum of the values in its subtree. Given the min-binary tree, it is straightforward to find the matches of all elements in parallel in $\mathcal{O}(\log n)$ time. Naively, this results in $\Theta(n \log n)$ work, but by applying a clever coarsening of the input into chunks of size $\Theta(\log n)$, they reduce the total work to $\mathcal{O}(n)$. We call this algorithm *BSV*, and due to the complexity of the algorithm, it had not been implemented prior to our work. Instead, Blelloch, Shun, and Zhao [26, 123, 124] implemented a heuristic algorithm that also builds on the min-binary tree but sidesteps the complicated input coarsening step. We call this algorithm *BSZ*. In [123], the authors note that the heuristic is surprisingly effective:

> *"... we note that the ANSV portion takes less than 2% of the total time even though it is an $\mathcal{O}(n \log n)$ work algorithm"*

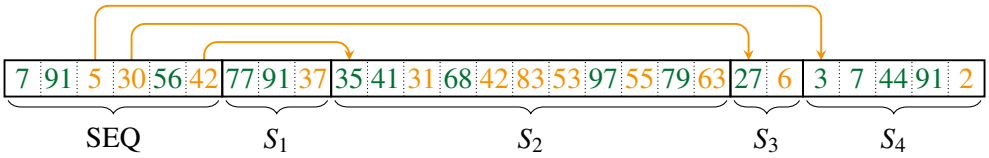
In manuscript 11, we address this gap between theory and practice by analyzing the heuristic algorithm of Blelloch, Shun, and Zhao [26, 123, 124], proving that it is work-efficient as stated in Theorem 15, thereby justifying its strong empirical performance. Second, we present the first implementation of the work-efficient BSV algorithm and demonstrate that it is practically viable and performs competitively with existing methods.

Theorem 15 *(Manuscript 11 Theorem 1) For an input of size n and any integer hyperparameter $1 \leq k \leq n$, the BSZ algorithm achieves $\mathcal{O}\left(n\left(1+\frac{\log n}{k}\right)\right)$ work and $\mathcal{O}\left(k(1+\log \frac{n}{k})\right)$ span.*

The span overhead is only significant when the number of processors is $\Omega(n/\log n)$.

The Heuristic Algorithm

We describe the heuristic algorithm by Blelloch et al. [26, 123, 124] in terms of a parameter k; originally, only $k = \Theta(\log n)$ was considered. The input is divided into $\lceil n/k \rceil$ blocks of k values, and all matches within each block are found sequentially. These are called *local matches*. Then, for each block, the unmatched elements form an increasing sequence whose values are matched using the min-binary tree. These are called *nonlocal matches*. Instead of doing so in parallel, the critical part of the heuristic is to do so sequentially and in reverse order (starting with the largest value),



(a) The recursive algorithm first finds local matches for 7, 91, and 56 among the first $k = 6$ values. The remaining three values, 5, 30, and 42, have nonlocal matches, indicated by arrows, which are found using the min-binary tree. These values partition the remaining input into four independent subproblems, S_1, S_2, S_3, and S_4, which are solved recursively.

(b) The heuristic algorithm partitions the input array into blocks of exactly k elements, with the final block possibly shorter than the others. Note how the nonlocal matches always form an increasing sequence in each block.

Figure 3.1: (Manuscript 11) The behavior of the heuristic and recursive algorithm on the same input. All elements are colored green and orange, respectively, depending on whether they find a local or nonlocal match.

always starting the search for the next match where the previous match was found. Figure 3.1b shows an example of the behavior of the heuristic algorithm. To show that this decreases the work to linear, with a modest increase in its span, we compare its performance to a novel recursive algorithm. We stress that the recursive algorithm is not implemented but is entirely used for the analysis.

The Recursive Algorithm

The recursive algorithm is also defined in terms of a parameter k. For the first block, the recursive algorithm behaves similarly to the heuristic algorithm, as it finds local matches and then nonlocal matches sequentially from right-to-left using the min-binary tree. The nonlocal matches partition the remaining input into at most $k+1$ disjoint and independent subproblems that are solved recursively. Figure 3.1a shows an example of the behavior of the recursive algorithm, and highlights for all elements whether they eventually find a local or nonlocal match. The work of the recursive algorithm is bounded by $\mathcal{O}(T(n))$, where $T(n)$ is the recurrence:

$$T(n) = \begin{cases} k + \max\left(\sum_{i=1}^{k+1} T(n_i) + \sum_{i=1}^{k} \log(n_i)\right) & n > k \\ n & n \leq k, \end{cases}$$

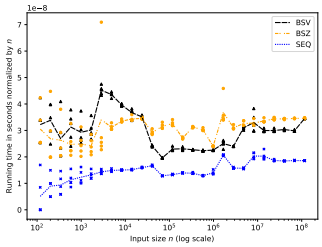
Analysis

We analyze the work of the heuristic algorithm W_{BSZ} in two steps. First, we show that its work is bounded by the work W_{REC} of the recursive algorithm, up to an additive

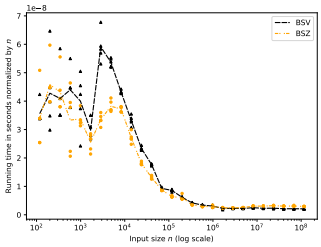
term. More precisely, we prove that $W_{BSZ} = \mathcal{O}\left(W_{REC} + n\left(1 + \frac{\log n}{k}\right)\right)$. To do so, for each block of k values, we compare the work each algorithm spends on nonlocal matches. In the second step, we show that $W_{REC} = \mathcal{O}(T(n)) = \mathcal{O}\left(n\left(1 + \frac{\log n}{k}\right)\right)$. This is done by proving via induction that $T(n) \leq 2n + \left(\frac{n}{k} - 1\right)\log n$. The span of the heuristic algorithm is straightforward to bound by $\mathcal{O}\left(k\left(1 + \log \frac{n}{k}\right)\right)$, by considering the worst-case scenario in which the matches of all elements in the first block are evenly distributed among the remaining elements.

Experiments

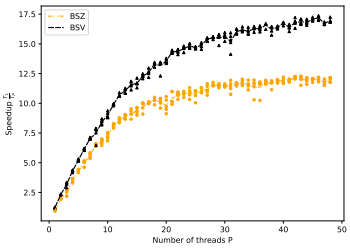
We present the first implementation of the original work-efficient algorithm by Berkman, Schieber and Vishkin [22]. Although the algorithm has been considered complex, we provide a concise $\sim$175 line C++ implementation, and demonstrate that in practice it performs comparably to the heuristic algorithm by Blelloch, Shun, and Zhao [26, 123, 124]. This is illustrated in Figure 3.2b. On a machine with 24 cores, we achieve up to 13.7 parallel speedup. We evaluate both algorithms on a variety of input types. Of particular interest is the RANDOMMERGE input, which is constructed by interleaving two sorted sequences and independently swapping each adjacent pair of elements with probability 0.5. This introduces randomness while preserving partial structure, resulting in a balanced mix of local and nonlocal matches. As shown in Figure 3.2a, the work of both BSV and BSZ is approximately twice that of the sequential stack-based implementation. Up to around $P = 12$, both the BSV and BSZ algorithms exhibit near-linear speedup, likely because the experiments were run on a dual-socket system with 12 cores per socket. Beyond 12 processors, the speedup continues to increase steadily up to 24 cores, after which it begins to taper off.



(a) Running time in seconds normalized by n for the SEQ, BSZ, and BSV algorithms with $P = 1$.



(b) Running time in seconds normalized by n for the BSZ and BSV algorithms with $P = 48$.



(c) The speedup T_1/T_P for the BSZ, and BSV algorithms for $P = 1,2,3,\ldots,48$.

Figure 3.2: (Manuscript 11) For all three plots, each dot corresponds to the running time in seconds of an algorithm on the RANDOMMERGE input. For each input size n, we repeat the experiment 5 times and draw a line through the average of these 5 runs. In each run we set the block size $k = 256 \log n$ to ensure $\Theta(n)$ work. For plots 3.2a and 3.2b we use a log scale and test inputs of size $n = 1.7^p$ for $p = 1,2,3,...$ and $n \leq 2^{27} = 134,217,728$. The running time is in seconds normalized by n (see the 1e-8 on the axis).

Chapter 4

Computational Geometry

In Chapter 2, we saw how geometric techniques can be instrumental in designing efficient persistent data structures. This chapter now shifts focus to present our contributions to the field of pure computational geometry. First, Section 4.1 details an efficient heuristic algorithm for outlier detection in two-dimensional point sets. This work, motivated by challenges in computational biology, is based on the principle of area-weighted convex hull peeling. Second, Section 4.2 resolves a recently posed open problem by showing that the contiguous art gallery problem can be solved in polynomial time. Our solution is a simple greedy algorithm that is iterated until an optimal set of guards is found. The proof of correctness, however, relies on a complex analysis of the geometric and combinatorial properties of the greedy strategy. Together, these works showcase two distinct facets of the field. The first leans toward practical application, while the second is entirely theoretical.

4.1 Area-Weighted Convex Hull Peeling

In this section, we present the contributions of Manuscript 10. The main result is an efficient real-RAM algorithm for area-weighted convex hull peeling in the plane. Given a set P of n points, the algorithm iteratively deletes the point with maximum *sensitivity*, defined for a point $p \in P$ as $\sigma(p) = \text{area}(\text{CH}(P)) - \text{area}(\text{CH}(P \setminus p))$, that is, the decrease in the area of the convex hull when p is removed. Figure 4.2 illustrates the sensitivity of a point. The removed point is always on the convex hull, and its removal is known as a *peel* [79, 122]. At each step, the algorithm peels the point with maximum sensitivity. The algorithm proceeds as follows:

1. Initialize the set P to contain all n input points.
2. Repeat the following n times:
    - Identify the point $u = \text{argmax}_{p \in P} \sigma(p)$.
    - Update $P \leftarrow P \setminus \{u\}$.

Depending on the application, the algorithm may be terminated after k iterations,

and may output the order in which points are removed from P and the area of the convex hull at each step.

The algorithm is based on the principle that outliers can be detected as points that cause a large decrease in the area of the convex hull when peeled. For any parameter $0 < k < n$, our algorithm identifies k outliers by the first k peels, and runs in $\mathcal{O}(n \log n)$ time. Using the area of the convex hull for outlier detection has been previously explored, primarily from the perspective of identifying the k points whose collective removal causes the largest decrease in area. We refer to such a removal as a *k-peel*. In contrast, our algorithm repeatedly performs k individual 1-peels, and can be viewed as an efficient *heuristic* for the *exact* computation of a single k-peel. Note that the exact k-peel always produces a convex hull whose area is less than or equal to that obtained by performing k sequential 1-peels. However, the phenomenon of *outlier masking* [98, 106, 107], in which nearby outliers obscure one another, can cause the difference in resulting area between the two approaches to become arbitrarily large. The computational complexity of existing exact algorithms and our heuristic is summarized in Table 4.1.

	Time	Space
Exact algorithms		
Eppstein [60, 62]	$\mathcal{O}\left(n^2 \log n + (n-k)^3\right)$	$\mathcal{O}\left(n \log n + (n-k)^3\right)$
Atanassov et al. [13]	$\mathcal{O}\left(n \log n + \binom{4k}{2k}(3k)^k n\right)$	—
Heuristic algorithms		
Sridhar and Svenning [127]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$

Table 4.1: Time and space bounds for exact and heuristic k-peeling algorithms. The space usage of Atanassov et al. was not stated.

We emphasize that for most values of n and k, our algorithm is the first practically viable algorithm. The exact methods are combinatorial in nature, whereas our algorithm is a greedy algorithm using dynamic convex hull data structures. To prove its efficiency we employ a novel amortized analysis. All algorithms generalize to other natural measures such as the perimeter of the convex hull.

Sensitivity

The sensitivity of a hull point u is determined by its own position, its neighbors on L_1, and its *active points*. A point v is said to be active for u if it contributes to $\sigma(u)$, which means that peeling u would cause v to appear on the convex hull. We denote the set of such points by $A(u)$, which consists precisely of the points on the second convex layer L_2 [43] that would advance to the first layer L_1 upon removal of u. Furthermore, only the points on L_1 have positive sensitivity, and the sensitivity of all points depends

solely on the set $L_1 \cup L_2$. It is therefore surprising that an $\mathcal{O}(n \log n)$ time algorithm exists. This is because a single peel can cause $\Omega(n)$ combinatorial changes to $A(u)$ for a single point, and the number of points whose sensitivity changes can also be $\Omega(n)$. Figure 4.1 illustrates this behavior and shows how the sensitivity of a point can be updated in time proportional to the number of changes to its active points. The core technical part of our analysis is Theorem 16, which bounds the total number of combinatorial changes to active points over the entire course of the peeling process.

Theorem 16 *(Manuscript 10) For any 2D convex hull peeling process on n points, the total number of times any point becomes active for another point is at most* $3n$.

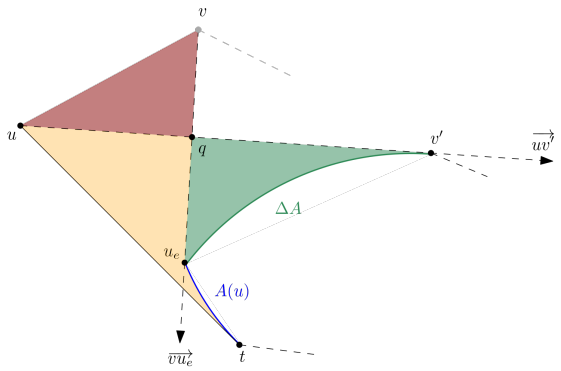


Figure 4.1: Before peeling v, the sensitivity $\sigma(u)$ is determined by the area of the polygon $\pentagon(t \circ u \circ v \circ u_e \circ A(u))$, where $\circ$ denotes the concatenation of vertices. After v is peeled, $\sigma(u)$ can be updated in $\mathcal{O}(|\Delta A|)$ time by subtracting the area of the red triangle $\triangle uvq$ and adding the area of the green polygon $\pentagon(u_e \circ q \circ v' \circ \Delta A)$. All points in ΔA then become active for u. The point q is not part of the input, but rather the intersection of the lines $\overrightarrow{vu_e}$ and $\overrightarrow{uv}$, where u_e is the point in $A(u)$ closest to v. If u were peeled instead of v, all points in $A(u)$ would transition from zero to positive sensitivity, and $|A(u)|$ may be $\Omega(n)$.

Description of the algorithm

Throughout the peeling process, we maintain three data structures. First, a max-priority queue [139] with the sensitivities of all points. Second, we maintain the two outer convex layers L_1 and L_2. We store them using balanced-binary trees to facilitate tangent queries from a point to the convex layer in $\mathcal{O}(\log n)$ time [88, 114]. This allows us to find the active points $A(u)$ for any $u \in L_1$ by issuing two tangent queries to L_2 from u's neighbors in L_1, followed by a walk along L_2, similar to [100]. Third, we store all remaining points $P \setminus \{L_1 \cup L_2\}$ in a dynamic convex hull data structure D_{CH} that supports deletions and extreme point (or tangent) queries in amortized $\mathcal{O}(\log n)$ time and initialization in $\mathcal{O}(n \log n)$ time [40, 77]. See Figure 4.2 for an example.

In each round, the priority queue determines which point is peeled. Then L_1, L_2, and D_{CH} are updated via tangent and extreme point queries to move points from L_2 to

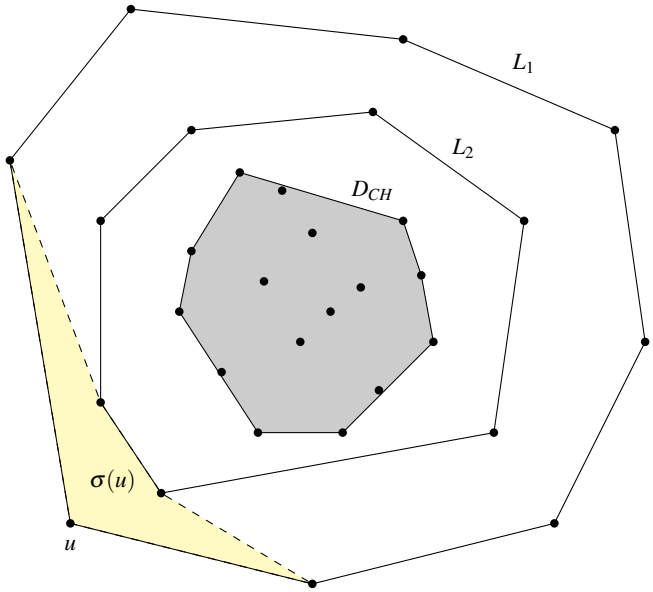


Figure 4.2: The data structures maintained during the peeling process. The algorithm maintains the two outer convex layers, L_1 and L_2, and a dynamic convex hull data structure D_{CH} for the remaining points. The yellow shaded area illustrates the sensitivity $\sigma(u)$ of point u, representing the decrease in convex hull area if u were removed. This area is defined by u, its neighbors on L_1, and its two active points on L_2.

L_1, and from D_{CH} to L_2. Though many points may be moved in a single iteration, each point moves monotonically from higher to lower *depth*. The depth of a point is defined as the number of convex layers enclosing it [43]. This allows us to charge $\mathcal{O}(\log n)$ time to each new point that appears in L_1 or L_2 after each peel, which suffices to cover the cost of updating L_1, L_2, and D_{CH}. The key technical challenge is to update the sensitivities of neighbors of the peeled point in time proportional to the number of new active points they gain. This is formalized in Lemma 17 and combined with the bound in Theorem 16. Figure 4.1 shows how the active points change when a neighboring point is peeled.

Lemma 17 *(Manuscript 10) Let (u,v) be points on the first layer. Consider a peel of v where δ_u new points become active points for u. Then the updated sensitivity $\sigma(u)$ can be computed in $\Theta(\delta_u + \log n)$ time, excluding the time to restore the second and first layer.*

Inspiration from Biology

Our motivation to study this problem arose from biology, specifically from geometric methods used in ecology to analyze biodiversity. Ecologists often represent species as points in a multidimensional trait space. This space is defined by characteristics such as body size, diet, or habitat preference. Biodiversity is then quantified using the area

or volume of the convex hull enclosing these points [47, 94]. In this setting, outliers correspond to species whose traits differ significantly from others, thus disproportionately enlarging the convex hull and potentially skewing ecological interpretations, including measures of biodiversity or niche variety.

Motivated by these practical biological scenarios, we considered a simplified but still challenging two-dimensional version of the problem, aiming to make statistical measures more robust to outliers. Given the size of ecological datasets, it was particularly important to develop a near-linear-time algorithm. Since our method allows efficient peeling of all points, it supports data exploration and improves interpretability. Plotting the area of the convex hull against the number of peeled points helps reveal the underlying structure of the data. Although peeling algorithms are well-established in computational geometry [61, 79, 100, 122], it was initially unclear whether an efficient solution would be possible for the area-weighted peeling problem. Our contribution lies in precisely formulating this biologically inspired geometric problem and proving that it admits an efficient solution.

Future Directions

Several directions remain open for future work on area-weighted convex hull peeling algorithms. First, a natural next step is to extend the algorithm to the three-dimensional setting. Two major challenges are adapting both dynamic convex hull data structures and the amortized analysis of Theorem 16 to three dimensions.

Second, in our current formulation, all points are peeled by performing successive 1-peels. A more accurate and robust approach would be to perform successive k-peels for values of $k > 1$, as each step would then remove a group of mutually reinforcing outliers, reducing the risk of outlier masking. However, even in the seemingly simple case of $k = 2$, it remains difficult to improve upon the naive strategy of repeatedly applying the exact algorithm of Atanassov et al. [13], which results in a total running time of $\Theta\big(n^2 \log n\big)$.

Third, it would be valuable to empirically evaluate the algorithm on real-world datasets and compare its performance with widely used methods such as Isolation Forest [98, 99]. Hybrid approaches could also be explored, where convex hull peeling is applied after clustering the data, using for example k-center clustering [75] or Local Convex Hulls [71], and outliers are identified within each cluster. The algorithm could prioritize which clusters to peel from by maintaining a global priority queue across the clusters. As always with outlier detection, the appropriate choice of method ultimately depends on some prior knowledge of the data and the type of outliers that one aims to detect.

4.2 The Contiguous Art Gallery Problem

The classical *art gallery problem*, posed by Klee in 1973 [110], asks for the minimum set of guards that can be placed in a polygon such that every point within the polygon is visible to at least one guard. Over the decades, many variants have been studied.

Most of these are known to be computationally hard, often being at least NP-hard. In this section, we consider the *contiguous art gallery problem*, a recently introduced variant in which the goal is to partition the boundary of a simple polygon P into a minimum number of contiguous polygonal chains, each of which must be visible to a guard located within P. We refer to each such visible portion of the boundary as a *visible interval*. Crucially, each guard may only cover a single contiguous portion of the boundary, and neither the guard locations nor the chain endpoints are required to coincide with polygon vertices. This variant was posed as an open problem by Shermer in 2024 as a candidate for a tractable restriction of the art gallery problem.

Results

In manuscript 9, we prove that the contiguous art gallery problem is solvable in polynomial time. In particular, we develop a simple greedy algorithm that always finds an optimal set of guards, as stated in Theorem 18, in the real-RAM model of computation.

Theorem 18 *(Manuscript 9 Theorem 1) There exists an algorithm for the contiguous art gallery problem for a simple polygon with n vertices that determines the optimal number of guards and their placement in* $\mathcal{O}\left(n^6 \log n\right)$ *arithmetic operations.*

We provide a C++ implementation of the algorithm, which is available [104] and demonstrates the simplicity of the approach. By bounding the bit complexity of the intermediate values computed by the algorithm, we show that the problem belongs to the class P. That is, it can be solved in polynomial time on a Turing machine [130], as stated in Theorem 19.

Theorem 19 *(Manuscript 9 Theorem 2) The contiguous art gallery problem for a simple polygon with vertices encoded as rational coordinates is in* P.

The fact that this contiguous variant admits an efficient algorithm is a surprising positive result, since most art gallery-type problems are known to be NP-hard [46] or even $\exists\mathbb{R}$-complete [41, 102, 119]. Our result contributes to clarifying the boundary between tractable and intractable variants of the art gallery problem. Biniaz et al.[24] and Robson et al.[117] independently proved the same main result, each using a different approach. The three approaches were subsequently unified in a joint publication [23].

Related Work

The art gallery problem has a long and well-studied history in computational geometry. The original version, in which guards must cover the entire polygon, is NP-hard when guards are restricted to vertices, as shown by Lee and Lin [96]. The edge-covering variant, in which only the boundary must be guarded, is also NP-hard for vertex guards, as shown by Laurentini [93]. The unrestricted versions of both the original and

edge-covering problems, where guards may be placed anywhere in simple polygons (possibly with holes), are $\exists\mathbb{R}$-complete [2]. In the tractable direction, Abrahamsen et al. [3] showed that the minimum star-shaped partition problem can be solved using $\mathcal{O}(n^{105})$ arithmetic operations. When the star-shaped regions are restricted to use only vertices of the input polygon, the complexity improves to $\mathcal{O}(n^7 \log n)$ operations [87]. The minimum star-shaped partition problem is NP-hard for polygons with holes [110]. A common theme across variants is that restricting guard positions and coverage regions to discrete locations, such as vertices of the input polygon, often reduces the complexity to membership in P. In contrast, for polygons with holes, many variants are at least NP-hard.

The contiguous variant is closely related to the *circular arc covering* problem, which asks for a minimum-size subset among m arcs that covers a circle, and can be solved in $\mathcal{O}(m \log m)$ time [95]. In our setting, the polygon boundary can be interpreted as a distorted circle, and an optimal guard cover corresponds to covering the boundary with visible intervals, each mapped to an arc on the circle. If the set of candidate visible intervals were finite and explicitly known, the Lee and Lee arc-covering algorithm could be applied directly. The difficulty lies in the fact that the number of candidate visible intervals may be infinite, as illustrated in Figure 4.3. Surprisingly, we show that the same greedy strategy used by Lee and Lee also yields an optimal solution in our setting. The strategy repeatedly selects the arc, or in our case the visible interval, that extends furthest along the boundary from the current position. In their setting, the arcs are given explicitly. In our geometric setting, each visible interval implicitly defines an arc.

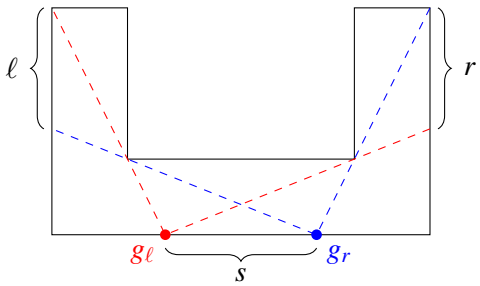


Figure 4.3: (Manuscript 9 Figure 2) A polygon where the blue guard g_r sees all of r and none of ℓ, and the red guard g_ℓ sees all of ℓ and none of r. Any guard placed along the segment s yields a trade-off between how much of ℓ and r it can see. This illustrates that, with unrestricted guard placements, it is non-trivial to extract a finite candidate set of visible intervals.

The Simple Greedy Algorithm

At the heart of our solution is the primitive operation GREEDYINTERVAL(x). Given a starting point x on the boundary of P, this operation computes the longest contiguous boundary interval $[x, y]$ in the clockwise direction that is visible from a guard point g

located in the interior of the polygon. That is, GREEDYINTERVAL extends forward from x along the boundary of the polygon until visibility is obstructed by the geometry. The operation returns the endpoint y of the visible interval, along with a guard position g that sees the entire interval $[x,y]$. A greedy interval can be computed in polynomial time using standard computational geometry procedures such as constructing visibility polygons [14, 59], computing polygon intersections [138], and finding common tangents [1].

Algorithm 1: (Manuscript 9 Algorithm 1)

Input: A simple polygon P with vertices $v_0, v_1, v_2, \ldots, v_n$
Output: An explicit solution to the contiguous art gallery problem for P

$x_0 \longleftarrow v_0$
for $i = 1$ **to** T **do**
  $x_i, g_i \longleftarrow GreedyInterval(x_{i-1}, P)$
$j \longleftarrow \max\{ j \leq T-2 \,|\, x_j \in (x_{T-1}, x_T] \}$
return $\{(x_{i-1}, x_i, g_i) \,|\, j < i \leq T\}$; `// The last segments/guards covering the boundary of` P

Algorithm 1 iterates the subroutine GREEDYINTERVAL T times, where T is a parameter, thereby traversing the boundary of the polygon. A single traversal proceeds as follows. The algorithm begins at a boundary point x_0 and performs GREEDYINTERVAL(x_0) to compute a maximal initial segment. The endpoint of this segment is denoted x_1. It then performs GREEDYINTERVAL(x_1) to cover the next segment, and continues in this manner. The traversal eventually reaches a point beyond x_0, indicating that the entire boundary is covered by the sequence of visible segments $[x_0,x_1],[x_1,x_2],\ldots,[x_{k-1},x_k]$. If the first pass uses k guards, we prove that this number is at most one more than the optimal, following a similar argument to that of Lee and Lee [95].

Our main technical result is the proof that $T = \Theta(n^4)$ revolutions suffice for Algorithm 1 to return an optimal solution. This bound on the number of iterations is the essential component in establishing the total runtime complexity stated in Theorem 18. The need for multiple revolutions arises because after the initial, possibly suboptimal pass, the algorithm may reduce the number of guards by one to reach optimality. To show this, we first characterize the combinatorial behavior of the endpoints generated by repeated applications of the GREEDYINTERVAL algorithm. We prove that if the sequence is *optimal*, then it remains optimal in further iterations. A sequence is optimal if it contains a consecutive set of guard intervals of optimal size that cover the boundary of P. Second, if the sequence is not optimal, then its endpoints can be restricted to certain fixed intervals of the boundary. Within each such interval, the endpoints move monotonically in the counterclockwise direction, as illustrated in Figure 4.4. We translate this behavior into a set of geometric conditions on the sequence of endpoints, which can be used to certify optimality. These conditions include a violation of monotonicity within an interval, repetition of an endpoint, or

the sequence passing more than $n+1$ vertices along the boundary of P.

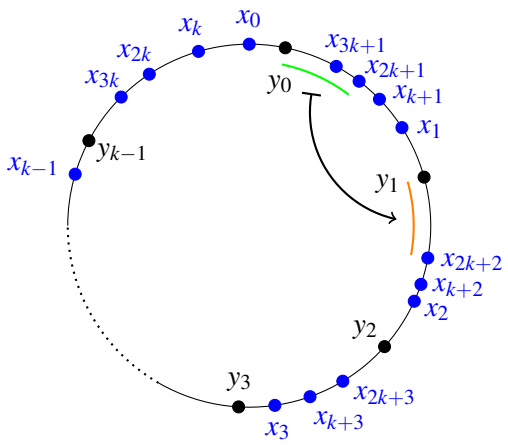


Figure 4.4: (Manuscript 9 Figure 18) The boundary of P visualized as a circle, where the y_i's indicate the endpoints of an optimal solution. The x_i's indicate a non-optimal sequence of endpoints generated by repeatedly running GREEDYINTERVAL, starting from x_0 and the last point generated being x_{3k+1}. If the sequence remains non-optimal, it maps points from $[y_0, x_{2k+1}]$ (green) to $[y_1, x_{2k+2}]$ (orange), and in the next revolution from $[y_0, x_{3k+1}]$ to $[y_1, x_{3k+2}]$.

Discussion

Empirical evidence suggests that the greedy algorithm is not only polynomial in theory but also highly efficient in practice. In extensive experiments on random polygons, the algorithm never required the worst-case bound of $\mathcal{O}(n^4)$ revolutions around the boundary. In fact, in more than two million random tests, an optimal solution was always obtained within *four* revolutions. Thus, we pose the conjecture that a constant number of revolutions suffices to find an optimal solution. Extending the algorithm to polygons with holes, or proving that this generalization is computationally hard, remains a natural open problem.

Part II

Publications

Chapter 5

External-Memory Priority Queues with Optimal Insertions

External-Memory Priority Queues with Optimal Insertions

Gerth Stølting Brodal
Aarhus University, Denmark

Michael T. Goodrich
University of California, Irvine, USA

John Iacono
Université libre de Bruxelles, Belgium

Jared Lo
University of Hawai'i at Mānoa, USA

Ulrich Meyer
Goethe University Frankfurt am Main, Germany

Victor Pagan
University of Hawai'i at Mānoa, USA

Nodari Sitchinava
University of Hawai'i at Mānoa, USA

Rolf Svenning
Aarhus University, Denmark

Abstract

We present an external-memory priority queue structure supporting Insert and DeleteMin with amortized $\mathcal{O}(1)$ and $\mathcal{O}(\lg N)$ comparisons, respectively, and amortized $\mathcal{O}\left(\frac{1}{B}\right)$ and $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ I/Os, respectively. Here, M is the size of the internal memory, B is the block size of I/Os between internal and external memory, and N is the number of elements in the priority queue just before an operation is performed. Previous external-memory priority queues required amortized $\mathcal{O}(\lg N)$ comparisons and $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ I/Os for both Insert and DeleteMin. The construction requires the minimal assumption $M \geq 2B$.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases priority queues, external memory, cache aware, amortized complexity

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.3

Funding *Gerth Stølting Brodal*: Supported by Independent Research Fund Denmark, grant 9131-00113B.
Michael T. Goodrich: Supported by NSF grant 2212129.
John Iacono: Research supported by the Fonds de la Recherche Scientifique — FNRS.
Jared Lo: Supported by NSF grant 2432018.
Ulrich Meyer: Supported by Deutsche Forschungsgemeinschaft (DFG) - ME 2088/5-2 (FOR 2975 - Algorithms, Dynamics, and Information Flow in Networks).
Victor Pagan: Supported by NSF grant 2432018.
Nodari Sitchinava: Supported by NSF grant 2432018.
Rolf Svenning: Supported by Independent Research Fund Denmark, grant 9131-00113B.

Acknowledgements Work initiated while attending the Third AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa, in part supported by the National Science Foundation under Grant No. 2452276.

1 Introduction

Priority queues are fundamental data structures with a host of applications. For example, algorithmic applications for priority queues range from classic results for sorting (such as heapsort) [13, 31, 32] and network optimization [19] to recent instance-optimal results for graph algorithms [22, 23]. In addition, priority queues have been developed for external-memory models [3, 6, 12, 18, 25, 26], ideal-cache models [4, 9], concurrent models [29], and RAM models [30]. Thus, it is desirable to design efficient priority queue data structures.

Throughout the paper, we assume a priority queue stores a multi-set of elements, where each element is a $\langle key, value\rangle$ pair, and the keys are from some totally ordered universe.

In the following, a comparison between two elements refers to the comparison of the two elements' keys.

Part of the reason priority queues are so broadly applicable is that they are defined in terms of two simple and useful operations:

- Insert(e) inserts an element e.
- DeleteMin() extracts and returns an element from the priority queue with minimum key. If several elements have equal keys, an arbitrary one of those is returned. This operation requires the priority queue to be non-empty before the operation.

Moreover, in internal memory, it is possible to design priority queues that hold N elements and achieve $\mathcal{O}(1)$ time for Insert and $\mathcal{O}(\lg N)$ time for DeleteMin, either in the worst case [14] or amortized [13, 31][1]. Unfortunately, prior to this work, we are not aware of any efficient data structures in external-memory models [1, 20, 21] that can match the performance of these classic internal-memory data structures.

Internal-memory priority queues

Williams introduced the binary heap in 1964 [32], which supports Insert and DeleteMin with $\mathcal{O}(\lg N)$ comparisons. Since then, many priority queues have been proposed; see, e.g., the survey by Brodal [7]. Vuillemin [31] introduced the binomial queue and demonstrated that it supports a sequence of N Insert and DeleteMin operations using a total of $\mathcal{O}(N \lg N)$ comparisons. Shortly after, Brown [13] gave a detailed analysis of binomial queues. Binomial queues support a sequence of I Insert and D DeleteMin operations in total $\mathcal{O}(I + D \lg I)$ comparisons. Carlson, Munro, and Poblete [14] showed how to achieve binomial queues supporting Insert with worst-case $\mathcal{O}(1)$ comparisons and DeleteMin with worst-case $\mathcal{O}(\lg N)$ comparisons. Fibonacci heaps, introduced by Fredman and Tarjan [19], achieve the same amortized performance as binomial queues for Insert and DeleteMin operations. Additionally, they support the DecreaseKey operation in amortized $\mathcal{O}(1)$ time and comparisons, allowing the key of an element to be replaced with a smaller key. Relaxed heaps, proposed by Driscoll, Gabow, Shrairman, and Tarjan [16], matched these bounds in the worst case.

External-memory model and priority queues

Aggarwal and Vitter [1] introduced the external-memory model as a model of computation focusing on the communication between two levels of memory: an *internal memory* of size M and an infinite *external memory*, where an *I/O* transfers a block of B consecutive memory cells between the two levels of memory. The *I/O cost* of an algorithm is the number of I/Os the algorithm performs. The minimal assumption is that $B \geq 1$ and $M \geq 2B$.

Aggarwal and Vitter proved that comparison-based sorting of N elements in the external-memory model can be done with $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os, and proved a matching lower bound for comparison-based sorting. Since sorting can be performed using a priority queue by performing N Insert operations followed by N DeleteMin operations, the amortized I/O cost of either Insert or DeleteMin must be $\Omega\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os. Arge [3], Kumar and Schwabe [27], and Fadel, Jakobsen, Katajainen, and Teuhola [18] presented external-memory priority queues supporting Insert and DeleteMin in amortized $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$

[1] $\lg x$ denotes the binary logarithm of x.

I/Os, assuming $M \geq 2B$. Brengel, Crauser, Ferragina and Meyer [6] presented a simple external-memory priority queue denoted an *array heap* with matching I/O bounds, assuming $B \log_{M/B} \frac{N}{B} \leq M$. Brodal and Katajainen [12] achieved a matching worst-case I/O cost. Since the number of DeleteMin operations is always no more than the number of Insert operation, we can charge the deletion cost to the insertions, i.e., we can restate the amortized costs to be $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os for Insert and $\mathcal{O}(1/B)$ I/Os for DeleteMin (or even zero). The question we address in this paper is whether we can swap the two I/O costs, i.e., achieve amortized $\mathcal{O}(1/B)$ I/Os for Insert and $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os for DeleteMin, which is relevant when the number of Insert operations is asymptotically larger than the number of DeleteMin operations, i.e., not all elements inserted into the priority queue are eventually also extracted. Table 1 summarizes previous results, where we let $\mathrm{Sort}(N) = \frac{N}{B} \log_{M/B} \frac{N}{B}$; hence, $\frac{1}{N}\mathrm{Sort}(N) = \frac{1}{B} \log_{M/B} \frac{N}{B}$.

Frigo, Leiserson, Prokop, and Ramachandran [20, 21] introduced the *ideal-cache* model by augmenting the external-memory model with an optimal offline paging mechanism. In this model, instead of having each algorithm explicitly transfer data between the two levels of memory, the data transfers are assumed to be performed by an optimal offline paging mechanism. While this might seem like a strong assumption, they demonstrated that under reasonable resource augmentation assumptions, a more realistic paging mechanism, e.g., the one implementing *least recently used (LRU)* replacement policy, is competitive with an optimal one. Consequently, ideal-cache model allows the design of so-called *cache-oblivious* algorithms – algorithms that are oblivious to the parameters M and B. Frigo et al. [20, 21] presented cache-oblivious sorting algorithms achieving optimal I/O cost, provided a *tall-cache assumption* of $M \geq B^2$. Brodal and Fagerberg [8] proved that the same bounds can be achieved under the weaker tall cache assumption $M \geq B^{1+\varepsilon}$ for any constant $\varepsilon > 0$, with a constant overhead of $\frac{1}{\varepsilon}$ in the I/O bounds. Brodal and Fagerberg [10] proved that this trade-off between the tall-cache assumption and the I/O cost is inherent to cache-oblivious algorithms. Cache-oblivious priority queues were presented by Arge, Bender, Demaine, Holland-Minkley, and Munro [4] and Brodal and Fagerberg [9] achieving $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os for Insert and DeleteMin, under the tall cache assumptions $M \geq B^2$ and $M \geq B^{1+\varepsilon}$, respectively.

In internal memory, several priority queues support Insert and DeleteMin in amortized $\mathcal{O}(\lg N)$ time, and additionally a DecreaseKey operation in amortized $\mathcal{O}(1)$ time (some also in the worst-case and Insert in $\mathcal{O}(1)$ time). A natural question is if it is possible to achieve similar results in external memory, i.e., N Insert and DeleteMin operations with $\mathcal{O}\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os and DecreaseKey with $\mathcal{O}\left(\frac{1}{B}\right)$ I/Os. Eenberg, Larsen, and Yu [17] proved a lower bound that this is not possible. An external-memory priority queue structure by Kumar and Schwabe [27] supports each of the three operations with amortized $\mathcal{O}\left(\frac{1}{B} \lg \frac{N}{B}\right)$ I/Os. Similar bounds were achieved cache-obliviously by Brodal, Fagerberg, Meyer, and Zeh [11] and Chowdhury and Ramachandran [15]. External-memory priority queues with improved DecreaseKey operations were presented by Iacono, Jacob, and Tsakalidis [25], who support Update (a combination of Insert and DecreaseKey) and DeleteMin in amortized $\mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right)$ and $\mathcal{O}\left(\left\lceil \frac{M^\varepsilon}{B} \log_{M/B} \frac{N}{B} \right\rceil \log_{M/B} \frac{N}{B}\right)$ I/Os, respectively, and by Jiang and Larsen [26], who support Insert, DeleteMin and DecreaseKey in expected amortized time $\mathcal{O}\left(\frac{1}{B} \lg \frac{N}{B} / \lg \lg N\right)$ I/Os, assuming $M > B \lg^{0.01} N$.

Table 1 Selected previous and new comparison and I/O bounds for priority queues. $\mathcal{O}_A$ denotes amortized bounds, whereas $\mathcal{O}$ denotes worst-case bounds.

	Comparisons		I/Os	
	Insert	DeleteMin	Insert	DeleteMin
Internal memory				
Binary heap [32]	$\mathcal{O}(\lg N)$	$\mathcal{O}(\lg N)$		
Binomial queue [13, 31]	$\mathcal{O}_A(1)$	$\mathcal{O}_A(\lg N)$		
Binomial queue [14]	$\mathcal{O}(1)$	$\mathcal{O}(\lg N)$		
External memory				
Buffer tree [3]	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$
Buffered multiway heap [18, 27]				
Brodal-Katajainen [12]	$\mathcal{O}(\lg N)$	$\mathcal{O}(\lg N)$	$\mathcal{O}\left(\frac{1}{N}\text{Sort}(N)\right)$	$\mathcal{O}\left(\frac{1}{N}\text{Sort}(N)\right)$
New (Theorem 1)	$\mathcal{O}_A(1)$	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A\left(\frac{1}{B}\right)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$
Cache oblivious				
Arge et al. [4], Funnel-heap [9]	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A(\lg N)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$	$\mathcal{O}_A\left(\frac{1}{N}\text{Sort}(N)\right)$

Result

In this paper, we present an external-memory priority queue that is optimal both with respect to the amortized I/Os and comparisons performed, achieving the following result.

▶ **Theorem 1.** *There exists an external-memory priority queue supporting* Insert *with amortized* $\mathcal{O}(1)$ *comparisons and* $\mathcal{O}\left(\frac{1}{B}\right)$ *I/Os, and* DeleteMin *with amortized* $\mathcal{O}(\lg N)$ *comparisons and* $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ *I/Os, where* N *is the current number of elements in the priority queue. The space usage is* $\mathcal{O}\left(\frac{N}{B}\right)$ *blocks. The memory size only needs to satisfy the minimal requirement* $M \geq 2B$.

We achieve this result using what can intuitively be considered an element "juggling" scheme, where we maintain three types of data structures—two in internal memory and one in external memory, transferring elements between them as needed. In a nutshell, our result is obtained by maintaining an internal-memory priority queue with the $\mathcal{O}(M)$ smallest elements supporting Insert and DeleteMin with $\mathcal{O}(1)$ and $\mathcal{O}(\lg M)$ comparisons. We also maintain an insert-buffer in internal memory as a buffer between the internal-memory priority queue and an external-memory structure comprising a forest of multi-way heaps. Specifically, in external memory, we store multi-way heap structures with buffers at the nodes, somewhat similar to an approach by Fadel, Jakobsen, Katajainen, and Teuhola [18] but adapted to support the insertion of batches of $\mathcal{O}(M)$ elements using $\mathcal{O}\left(\frac{M}{B}\right)$ I/Os. Whereas nodes in this previous construction store sorted buffers, in our construction, we maintain the forest of heaps in external memory to use partially sorted buffers (*semi-sorted* at internal nodes and *lazy semi-sorted* at the leaves) to avoid insertions requiring amortized $\Omega(\lg M)$ comparisons.

2 External-memory priority queue

In Sections 3–7, we describe the details of an external-memory priority queue achieving Lemma 2 below, where we bound the total number of comparisons and I/Os performed by a sequence of priority queue operations in terms of the total numbers of Insert and DeleteMin operations performed, denoted I and D, respectively. In Section 8, we then apply global rebuilding to make the bounds depend on the current number of elements in the priority queue, achieving the amortized bounds in Theorem 1.

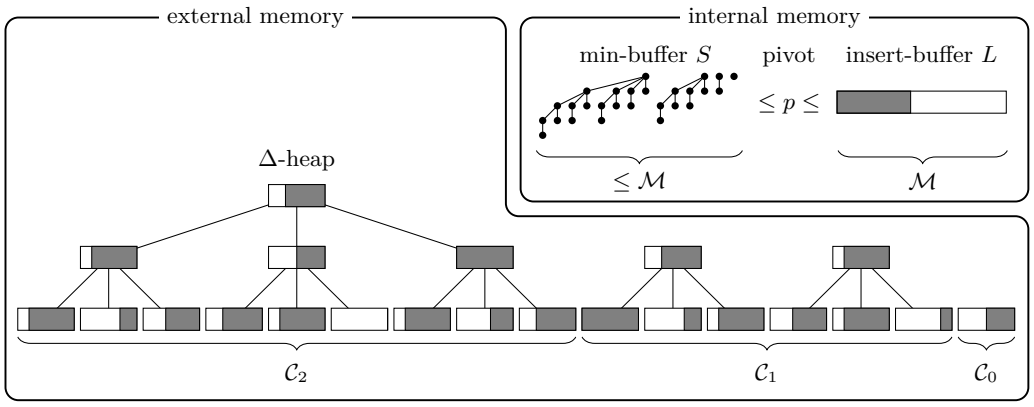


Figure 1 External-memory priority queue. All rectangles are buffers of capacity $\mathcal{M}$, where the gray area illustrates elements in buffers.

▶ **Lemma 2.** *There exists an external-memory priority queue supporting a sequence of I* Insert *and D* DeleteMin *operations, using a total of $\mathcal{O}(I + D \lg I)$ comparisons and $\mathcal{O}\left(\frac{1}{B}\left(I + D \log_{M/B} \frac{I}{B}\right)\right)$ I/Os, assuming $M \geq 2B$.*

Our priority queue consists of an internal-memory part and an external-memory part; see Figure 1 for an overview. Each element is stored exactly once in our data structure. In internal memory we store a *min-buffer*, a *pivot* element, and an *insert-buffer*, where the elements in the min-buffer are all smaller than or equal to the pivot, and the elements in the insert-buffer and external memory are all larger than or equal to the pivot element. By default, Insert and DeleteMin are performed in internal memory without performing any I/Os. If internal memory contains too many elements, $\Theta(M)$ elements are moved from the insert-buffer to the external part. To perform DeleteMin when the min-buffer is empty, $\Theta(M)$ smallest elements from external memory are moved to internal memory. We call moving $\Theta(M)$ elements between internal and external memory a *transfer*. In the subsequent sections, we describe the two parts in detail. We let $\mathcal{M}$ be a parameter for our construction, such that $\mathcal{M}$ is even, $\mathcal{M}$ is divisible by B, and $\mathcal{M} = \Theta(M)$. We choose $\mathcal{M}$ such that the $\mathcal{O}(\mathcal{M})$ space internal-memory data structure in Section 3 fits into internal memory. To be able to choose $\mathcal{M}$, M is required to be sufficiently large and the internal memory to be able to hold a sufficiently large number of blocks of size B. If $M = \mathcal{O}(1)$, then an internal-memory data structure like a binomial queue already achieves the result of Lemma 2, and if the number of blocks fitting in internal memory is too small we can simulate a smaller block size with a constant blow up in the I/O complexity. I.e., in the following we w.l.o.g. can assume that M and M/B are sufficiently large to be able to choose $\mathcal{M}$.

3 The internal-memory part

The internal-memory part consists of a *min-buffer* S, a *pivot* element p, and an *insert-buffer* L. The pivot is larger than or equal to any element in the min-buffer, and smaller than or equal to any element in the insert-buffer and external memory. The insert-buffer is an unordered array of at most $\mathcal{M}$ elements. The min-buffer stores the overall at most $\mathcal{M}$ smallest elements in the priority queue, and is implemented using an internal-memory linear-space priority queue supporting Insert in amortized $\mathcal{O}(1)$ time and DeleteMin in amortized $\mathcal{O}(\lg n)$ time, where n is the number of elements in the min-buffer. The min-buffer can, e.g., be

implemented using a binomial queue [31] or the implicit post-order heap by Harvey and Zatloukal [24]. The total space for the internal part is $\mathcal{O}(\mathcal{M})$.

INSERT(e) first compares e to the pivot p. If $e \leq p$, e is inserted in the min-buffer using the min-buffer's INSERT operation. Otherwise, e is appended to the insert-buffer. If the min-buffer overflows, i.e., gets size $\mathcal{M}+1$, we find a new pivot p' by applying the linear-time internal-memory median finding algorithm by Blum, Floyd, Pratt, Rivest, and Tarjan [5] to the elements in the min-buffer. The median, i.e., the $\left(\frac{\mathcal{M}}{2}+1\right)$th smallest element, becomes the new pivot p', the $\frac{\mathcal{M}}{2}$ larger elements and the old pivot p are appended to the insert-buffer, and the $\frac{\mathcal{M}}{2}$ smaller elements are inserted into a new empty min-buffer structure using its INSERT operations. If the insert-buffer overflows, i.e., gets size $> \mathcal{M}$ (due to the insertion of a single element or the old pivot p and $\frac{\mathcal{M}}{2}$ elements being moved from the min-buffer to the insert-buffer), we transfer $\mathcal{M}$ elements from the insert-buffer to the external part (see Section 4).

For DELETEMIN, if the min-buffer is non-empty, the smallest element is extracted from the min-buffer using the min-buffer's DELETEMIN operation and returned. Otherwise, the min-buffer is empty and the pivot p is the smallest element to be returned. To establish a new pivot p' and min-buffer, the $\frac{\mathcal{M}}{2}$ smallest elements are extracted from external memory (see Section 4) and transferred to the insert-buffer (except when the external part is empty). This ensures that the $\frac{\mathcal{M}}{2}$ smallest elements in the insert-buffer are now smaller than or equal to all elements in the external part. The $\frac{\mathcal{M}}{2}$th smallest element in the insert-buffer is selected as the new pivot p' in linear time using the internal-memory selection algorithm [5]. If the insert-buffer contains $< \frac{\mathcal{M}}{2}$ elements, we set the pivot p to be $+\infty$. The at most $\frac{\mathcal{M}}{2}-1$ elements smaller than or equal to the new pivot p' in the insert-buffer are deleted from the insert-buffer and inserted into the min-buffer structure using its INSERT operation. The old pivot p is returned as the extracted minimum.

Note that a transfer either moves $\mathcal{M}$ elements from the internal to the external part, or $\frac{\mathcal{M}}{2}$ elements from the external to the internal part, i.e., the external memory always contains a multiple of $\frac{\mathcal{M}}{2}$ elements. Note also that the size of the insert-buffer is unchanged during DELETEMIN, provided that there are elements in external memory. Otherwise, the number of elements in the insert-buffer decreases by $\frac{\mathcal{M}}{2}$.

Initially, the pivot $p = +\infty$, such that all elements are inserted into the min-buffer until the priority queue contains $\mathcal{M}+1$ elements, where a real element is selected as the pivot and $\frac{\mathcal{M}}{2}$ elements are moved to the insertion-buffer and a new min-buffer is created for the $\frac{\mathcal{M}}{2}$ smallest elements using repeated insertions.

4 The external-memory part

The external-memory part supports the insertion of a batch of $\mathcal{M}$ elements and the extraction of the $\mathcal{M}/2$ smallest elements. It consists of a set of Δ-*heaps*, where $\Delta = \frac{\mathcal{M}}{B} \geq 2$. A Δ-heap with height h is a tree where all leaves have depth h (the root having depth zero), and all non-leaves have exactly Δ children. Each node contains a buffer storing up to $\mathcal{M}$ elements. The elements must satisfy *extended heap order*, i.e., the elements in the buffer of a node u are smaller than or equal to any element in the buffers of the children of u. A Δ-heap is a generalization of a binary heap [32] and is very similar to the external-memory priority queue of Fadel, Jakobsen, Katajainen and Teuhola [18], except that we do not require buffers to be sorted (this would not be possible when insertions only perform amortized $\mathcal{O}(1)$ comparisons.) The buffers in a Δ-heap store elements in various degrees of sortedness to be able to achieve the stated insertion cost: non-leaves are *semi-sorted*, whereas leaves are *lazy semi-sorted* as

discussed in Sections 5 and 6, respectively.

The Δ-heaps are maintained as a sequence of collections $\mathcal{C}_0, \ldots, \mathcal{C}_H$, where $\mathcal{C}_h$ contains all Δ-heaps with height h. We maintain the invariant (**I1**) that $|\mathcal{C}_h| < \Delta$ and that the total number of leaves in the Δ-heaps is $\leq \frac{I}{\mathcal{M}}$ (Lemma 7), where I is the total number of insertions. This invariant ensures that the maximum height H of a Δ-heap is $\leq \lfloor \log_\Delta \frac{I}{\mathcal{M}} \rfloor$. The buffers of all nodes must satisfy the following invariant (**I2**): The buffer of a node u stores at most $\mathcal{M}$ elements. If no descendant of u stores elements, there is no lower bound on the buffer size of u. Otherwise, the buffer of u contains at least $\frac{\mathcal{M}}{2}$ elements. Together with the extended heap order, this invariant implies that the $\frac{\mathcal{M}}{2}$ smallest elements in a Δ-heap are stored in the root buffer, except if the Δ-heap contains fewer than $\frac{\mathcal{M}}{2}$ elements, in which case all elements are stored in the root buffer.

When $\mathcal{M}$ elements are transferred from the internal to the external part, the $\mathcal{M}$ elements become a single node Δ-heap with height zero that is added to collection $\mathcal{C}_0$. If the collection is full, that is, $|\mathcal{C}_0| = \Delta$, then a new root r is created for a Δ-heap of height one with each leaf of $\mathcal{C}_0$ as its children. Invariant **I2** is established for r by recursively *pulling* the $\frac{\mathcal{M}}{2}$ smallest elements from its children and inserting them in the buffer (the pull operation is described below). This leaves $\mathcal{C}_0$ empty, and r is promoted to $\mathcal{C}_1$. As in a Δ-ary number system, this promotion may propagate up through the collections, repeatedly combining Δ Δ-heaps with height h to a Δ-heap with height $h+1$.

For a node with $< \frac{\mathcal{M}}{2}$ elements in its buffer, a *pull* operation moves $\frac{\mathcal{M}}{2}$ elements from its children to the buffer of the node while preserving extended heap order, using Lemma 5 in Section 5. Note that this will make the buffer contain at least $\frac{\mathcal{M}}{2}$ elements and less than $\mathcal{M}$. If a child buffer gets size $< \frac{\mathcal{M}}{2}$, we recursively apply the pull operation to the child, provided that not all subtrees below the child are exhausted.

To extract the $\frac{\mathcal{M}}{2}$ smallest elements from the external part, first, for each collection $\mathcal{C}_i$, the $\frac{\mathcal{M}}{2}$ smallest elements are found by considering the elements in the buffers of the roots by applying Lemma 5 to the roots of the collection. As the elements are just identified but not pulled from the roots, we call this a *virtual pull*. Second, the $\frac{\mathcal{M}}{2}$ smallest elements among the $(H+1)\frac{\mathcal{M}}{2}$ candidates from the virtual pulls are found by applying a linear-time selection algorithm, Corollary 3 below. These elements are removed from their respective roots and transferred to the internal part. For each root buffer now containing $< \frac{\mathcal{M}}{2}$ elements, we recursively pull $\frac{\mathcal{M}}{2}$ elements from its children, to the extent possible.

Note that the total number of elements in external memory is always a multiple of $\frac{\mathcal{M}}{2}$, but this is not necessarily true for each Δ-heap due to the extractions of the smallest $\frac{\mathcal{M}}{2}$ elements across all Δ-heaps.

5 Semi-sorted lists

Given two sequences X and Y, we say $X \preceq Y$, if for every $x \in X$ and $y \in Y$: $x \leq y$. Note that $X \preceq Y$ implies no specific order within each individual sequence X or Y.

A buffer in external memory is a linked list of blocks $b_1, b_2, \ldots, b_\delta$, where each block contains B elements, except possibly the first and last blocks, which contain $\leq B$ elements. In the following, we simply denote such a linked list as a *list*. We define a list to be *semi-sorted* if $b_i \preceq b_{i+1}$ for every $1 \leq i < \delta$. All buffers of non-leaf nodes in a Δ-heap are stored as semi-sorted lists. The following are useful tools for working with semi-sorted buffers.

Blum, Floyd, Pratt, Rivest, and Tarjan [5] proved that the kth smallest element in a list with N elements can be found using $\mathcal{O}(N)$ comparisons using a double recursion making repeated use of scanning lists. Analyzed in the external-memory model, this immediately

implies that selection can be performed in a linear number of I/Os, and we have the following corollary.

▶ **Corollary 3** (Blum et al. [5]). *Given a list of N elements and an integer $1 \leq k \leq N$, the list can be partitioned into two lists X and Y, with $|X| = k$ and $X \preceq Y$, using $\mathcal{O}(N)$ comparisons and $\mathcal{O}\left(\frac{N}{B}\right)$ I/Os.*

▶ **Lemma 4** (Constructing semi-sorted buffer). *An unsorted list of $B\delta \leq \mathcal{M}$ elements can be converted into a semi-sorted list using $\mathcal{O}(B\delta \lg \delta)$ comparisons and $\mathcal{O}(\delta)$ I/Os.*

Proof. Recursively apply [5] to partition an unsorted list with δB elements into two subproblems with $\lceil \delta/2 \rceil B$ and $\lfloor \delta/2 \rfloor B$ elements, until the subproblems are of size exactly B. The result follows since there are $\lceil \lg \delta \rceil$ levels of recursion with a linear number of comparisons per level, and we only perform I/Os to read the initial list into internal memory and to write the final list to external memory, since the problem fits in internal memory. ◀

▶ **Lemma 5** (Pulling from semi-sorted lists). *Given at most $\Delta = \frac{\mathcal{M}}{B}$ semi-sorted lists, a semi-sorted list of the $\frac{\mathcal{M}}{2}$ smallest elements of their union can be computed in $\mathcal{O}(\mathcal{M} \lg \Delta)$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os.*

Proof. Read the first block from each semi-sorted list into internal memory and insert the largest element of each block into a binary min-heap Q [32]. Then, delete the smallest element from Q, read the next block from the corresponding semi-sorted list, and insert the largest element of this block into Q. Let X denote the elements in blocks whose maximums have been deleted from Q, and Y the elements in blocks whose maximums are in Q. Repeat this process of reading blocks until $|X| \geq \frac{\mathcal{M}}{2}$. In total, $\Theta\left(\frac{\mathcal{M}}{B}\right)$ blocks are read, where the maximum is computed of each block and each block requires $\mathcal{O}(1)$ heap operations on Q, where $|Q| \leq \Delta$. This initial phase requires a total of $\mathcal{O}\left(\mathcal{M} + \frac{\mathcal{M}}{B} \cdot \lg \Delta\right)$ comparisons and $\Theta\left(\frac{\mathcal{M}}{B}\right)$ I/Os.

Let x be the last element deleted from Q. We have $\max(X) \leq x$, since when the lists are semi-sorted, elements will be extracted in non-decreasing order from Q. From each list, we have a block represented in Q with maximum $\geq x$, so all remaining unread elements in the lists are also $\geq x$. Thus, the $\frac{\mathcal{M}}{2}$ smallest elements can be found by a single partition of $X \cup Y$ using $\mathcal{O}(\mathcal{M})$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os per Corollary 3. They can then be converted to a semi-sorted list using additional $\mathcal{O}(\mathcal{M} \lg \Delta)$ comparisons per Lemma 4. ◀

6 Lazy semi-sorted lists

Since I insertions can create $\Theta(I/\mathcal{M})$ leaf buffers of size $\mathcal{M}$, they cannot all be semi-sorted, as that would require $\Theta(I \lg \Delta)$ comparisons. The lower bound of $\Omega(\mathcal{M} \lg \Delta)$ comparisons for the multiple selection problem for each buffer is due to Dobkin and Munro [2, Theorem 1]. Instead, a leaf buffer with $\mathcal{M}$ elements is stored as a *lazy semi-sorted* list, see Figure 2. A lazy semi-sorted list stores the $\mathcal{M}$ elements as a sequence of *chunks* $c_1 \preceq c_2 \preceq \cdots \preceq c_d$, where $d = \left\lceil \lg \frac{\mathcal{M}}{B} \right\rceil + 1$ and, initially, the elements inside a chunk are stored in arbitrary order. Chunk c_1 stores B elements, c_i stores $B2^{i-2}$ elements for $2 \leq i < d$, and c_d stores the largest $\mathcal{M} - B2^{d-2}$ elements. If each chunk is converted into a semi-sorted list, and all chunks are concatenated, we have a semi-sorted list for the buffer. The basic idea is to exploit that a Δ-heap always accesses the blocks of a semi-sorted buffer left-to-right, implying that we can delay expanding a chunk into a semi-sorted sequence of blocks using Lemma 4 until the first block of the (semi-sorted version of the) chunk is accessed, hence the name lazy semi-sorted.

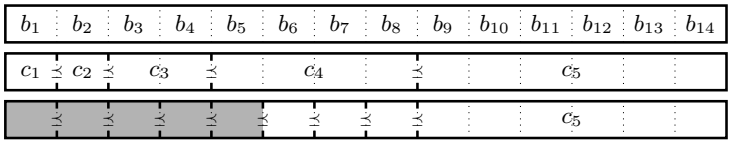


Figure 2 A lazy semi-sorted list. Top: Initial unsorted list of blocks $b_1, \ldots, b_{\mathcal{M}/B}$. Middle: Initial chunks $c_1 \preceq \cdots \preceq c_5$. Bottom: The state after the first five blocks $b_1, \ldots, b_5$ in the semi-sorted list have been accessed, i.e., the chunks $c_1, \ldots, c_4$ have been made into semi-sorted sublists.

The initial partitioning of a buffer into chunks is done by recursively partitioning the half with the smaller elements using Corollary 3.

▶ **Lemma 6** (Lazy access cost). *Initializing a lazy semi-sorted list of $\mathcal{M}$ elements requires $\mathcal{O}(\mathcal{M})$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os. The cost of accessing the first $k \leq \frac{\mathcal{M}}{B}$ blocks of the lazy semi-sorted list requires $\mathcal{O}(Bk \lg k)$ comparisons and $\mathcal{O}(k)$ I/Os.*

Proof. To initialize the lazy semi-sorted list, all $\mathcal{M}$ elements are first read into internal memory using $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os. No further I/Os are performed, except for eventually writing the output to external memory. The initial chunks, with elements in arbitrary order, are constructed one at a time by repeatedly applying Corollary 3 to find the appropriate number of largest elements. By constructing the chunks in the order $c_d, c_{d-1}, \ldots c_0$ from the largest to the smallest, the number of remaining elements decreases geometrically, leading to a total of $\mathcal{O}(\mathcal{M})$ comparisons.

If the k first blocks of elements have been accessed, then chunks $c_1, \ldots, c_{\lceil \lg k \rceil + 1}$ have been semi-sorted, i.e., fewer than $2k$ blocks. Note that c_1 and c_2 only consist of a single block, i.e., they are already semi-sorted at initialization. By applying Lemma 4, semi-sorting the remaining $\lceil \lg k \rceil - 1$ accessed chunks requires $\mathcal{O}\left(\sum_{i=1}^{\lceil \lg k \rceil - 1} B 2^i \lg 2^i\right) = \mathcal{O}(Bk \lg k)$ comparisons and $\mathcal{O}\left(\sum_{i=1}^{\lceil \lg k \rceil - 1} 2^i\right) = \mathcal{O}(k)$ I/Os. ◀

The idea of lazy semi-sorted buffers is that they allow us to charge the non-linear comparison cost of constructing a semi-sorted buffer to the pull operations on the buffer, in particular, the pull operations triggered by deletions.

7 Analysis

When Insert and DeleteMin can be performed entirely within the internal part, no I/Os are performed. The number of comparisons is $\mathcal{O}(1)$ for Insert and $\mathcal{O}(\lg \min(\mathcal{M}, I))$ for DeleteMin, plus the cost of moving elements from the min-buffer to the insert-buffer. If an insertion causes the min-buffer to overflow, we spend $\mathcal{O}(\mathcal{M})$ comparisons moving elements to the insert-buffer and building a new min-buffer. This can only happen once every $\frac{\mathcal{M}}{2}$ insertion, so the total cost for this is $\mathcal{O}(I)$ comparisons and no I/Os.

A transfer either moves $\mathcal{M}$ elements from the internal to the external part, or $\frac{\mathcal{M}}{2}$ elements from the external to the internal part. The following lemma bounds the number of transfers in each direction in terms of I and D.

▶ **Lemma 7** (Bounding transfers). *The number of transfers from the internal to the external part is $\leq \frac{I}{\mathcal{M}}$. The number of transfers from the external to the internal part is $\leq \frac{2D}{\mathcal{M}}$.*

Proof. We first consider the number of transfers from the internal to the external part. Let $\phi \geq 0$ be the number of elements in the min-buffer beyond the first $\frac{\mathcal{M}}{2}$ elements plus the

number of elements in the insert-buffer. Insertions cause ϕ to increase by at most one, and deletions do not cause ϕ to increase. A transfer to the external part decreases ϕ by $\mathcal{M}$, whereas a transfer to the internal part of $\frac{\mathcal{M}}{2}$ elements leaves ϕ unchanged, since the min-buffer gets size $\frac{\mathcal{M}}{2} - 1$ and the size of the insert-buffer does not change. It follows that ϕ only increases during the I insertions, where the total increase is $\leq I$, and each transfer to the external part decreases ϕ by $\mathcal{M}$, i.e., the number of transfers to the external part is at most $\frac{I}{\mathcal{M}}$.

The size of the min-buffer can only decrease below $\frac{\mathcal{M}}{2}$ due to a deletion, where it decreases by one. (When the size of the min-buffer decreases because half of the elements are moved to the insert-buffer, the min-buffer changes size from $\mathcal{M}+1$ to $\frac{\mathcal{M}}{2}$.) A transfer to the internal part occurs when the size of the min-buffer is zero before the deletion, but then the min-buffer has size $\frac{\mathcal{M}}{2} - 1$ after the deletion and the next $\frac{\mathcal{M}}{2} - 1$ deletions do not cause a transfer to the internal part. It follows that at most every $\frac{\mathcal{M}}{2}$ deletion can cause a transfer to the internal part. The first transfer to the internal part can only happen after a transfer to the external part has occurred, which first can happen after the pivot $p \neq +\infty$, where the size of the min-buffer is $\frac{\mathcal{M}}{2}$, i.e., there must have been at least $\frac{\mathcal{M}}{2}$ deletions before the first transfer to the internal part. It follows that the number of transfers to the internal part is at most $\frac{2D}{\mathcal{M}}$. ◀

The key part of the analysis is threefold. First, we bound the cost to restore invariants **I1** and **I2** whenever elements are removed from the internal nodes of the external part. Second, we bound the cost of *accessing* the lazy semi-sorted leaves. The access cost of a leaf refers to the cost of semi-sorting its chunks. Third, we bound the cost involved with transfers between internal and external memory. We define the *height* of a node in a Δ-heap to be the height of the subtree rooted at the node (leaves have height zero).

▶ **Lemma 8** (Total pull cost excluding leaf access). *The total number of pulls is $\mathcal{O}\big(\frac{I}{\mathcal{M}\Delta} + \frac{DH}{\mathcal{M}}\big)$ and their total cost is $\mathcal{O}\Big(I\frac{\lg\Delta}{\Delta} + D\lg\frac{I}{\mathcal{M}}\Big)$ comparisons and $\mathcal{O}\big(\frac{I}{\mathcal{M}} + \frac{DH}{B}\big)$ I/Os, excluding the cost of accessing the leaves.*

Proof. A pull at a node increases the height of $\frac{\mathcal{M}}{2}$ elements from the buffers of the children by one, except the last pull at a node that exhausts all subtrees at the children, where at most $\frac{\mathcal{M}}{2}$ elements have their height increased by one. By Lemma 7, at most $\frac{I}{\mathcal{M}}$ leaves are created, i.e., the maximum height of a Δ-heap is $H \leq \log_\Delta \frac{I}{\mathcal{M}}$. For a height h, $1 \leq h \leq H$, we bound the number of pulls P_h to nodes with height h by bounding the number of elements that could have been moved to a height $\geq h$ or transferred to the internal memory, the latter being bounded by D by Lemma 7. Since there are $\leq \frac{I}{\mathcal{M}}$ leaves, the number of nodes at height h is $\leq \frac{I}{\mathcal{M}\Delta^h}$, implying that the number of elements that could ever have been pulled to height h (and possibly further up) is at most $D + \mathcal{M}\sum_{j=h}^{H}\frac{I}{\mathcal{M}\Delta^j}$. Since each non-exhausting pull moves $\frac{\mathcal{M}}{2}$ elements one level up, and each node with height h can have one final pull exhausting all children, we get that the final number of pulls to height h is $P_h \leq \Big(D + \mathcal{M}\sum_{j=h}^{H}\frac{I}{\mathcal{M}\Delta^j}\Big) \,/\, \frac{\mathcal{M}}{2} + \frac{I}{\mathcal{M}\Delta^h} = \mathcal{O}\big(\frac{I}{\mathcal{M}\Delta^h} + \frac{D}{\mathcal{M}}\big)$, as $\Delta \geq 2$.

The total number of pulls is then $\sum_{h=1}^{H} P_h = \mathcal{O}\big(\frac{I}{\mathcal{M}\Delta} + \frac{DH}{\mathcal{M}}\big)$. Excluding the cost of accessing the leaves, we may treat each pull as being from semi-sorted lists. Applying Lemma 5, the total number of comparisons is $\mathcal{O}\big(\big(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\log_\Delta\frac{I}{\mathcal{M}}\big)\,\mathcal{M}\lg\Delta\big) = \mathcal{O}\Big(I\frac{\lg\Delta}{\Delta} + D\lg\frac{I}{\mathcal{M}}\Big)$ and the number of I/Os is $\mathcal{O}\big(\big(\frac{I}{\mathcal{M}\Delta} + \frac{DH}{\mathcal{M}}\big)\,\frac{\mathcal{M}}{B}\big) = \mathcal{O}\big(\frac{I}{\mathcal{M}} + \frac{DH}{B}\big)$ I/Os. ◀

▶ **Lemma 9** (Total leaf access cost). *The total cost of constructing the lazy semi-sorted leaves and accessing them is $\mathcal{O}(I + D\lg\Delta)$ comparisons and $\mathcal{O}\big(\frac{I}{B} + \frac{D}{B}\big)$ I/Os.*

Proof. As described in the proof of Lemma 8, the number of pulls from the leaves is $P_1 = \mathcal{O}\left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)$. Virtual pulls may, however, also access elements in the leaves. Since each transfer to the internal part causes one virtual pull from $\mathcal{C}_0$, the number of virtual pulls from the leaves is bounded by $\frac{2D}{\mathcal{M}}$ by Lemma 7. The total number of pulls and virtual pulls from the leaves is $\mathcal{O}\left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)$.

By Lemma 6, the cost for the initial construction of the at most $\frac{I}{\mathcal{M}}$ leaves as lazy semi-sorted lists is $\mathcal{O}(I)$ comparisons and $\mathcal{O}\left(\frac{I}{B}\right)$ I/Os. Since, by Lemma 5, each pull and virtual pull performs $\mathcal{O}(\mathcal{M} \lg \Delta)$ comparisons and accesses $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ blocks, we have, by Lemma 6, that at the leaves, the total number of comparisons is $\mathcal{O}\left(I + \left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)\mathcal{M} \lg \Delta\right) = \mathcal{O}(I + D \lg \Delta)$ comparisons and the total number of I/Os is $\mathcal{O}\left(\frac{I}{B} + \left(\frac{I}{\mathcal{M}\Delta} + \frac{D}{\mathcal{M}}\right)\frac{\mathcal{M}}{B}\right) = \mathcal{O}\left(\frac{I}{B} + \frac{D}{B}\right)$ I/Os. ◀

▶ **Lemma 10** (Cost of transfers excluding pulls and leaf accessing)**.** *The total cost of transfers between the internal and external part is $\mathcal{O}\left(D \lg \frac{I}{\mathcal{M}}\right)$ comparisons and $\mathcal{O}\left(\frac{I}{B} + \frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)$ I/Os, excluding the cost of pulling to restore the invariants and to access the leaves.*

Proof. By Lemma 7, there are at most $\frac{I}{\mathcal{M}}$ transfers to the external part, each moving $\mathcal{M}$ elements from the input-buffer to a new leaf, with no comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B}\right)$ I/Os.

By Lemma 7, there are at most $\frac{2D}{\mathcal{M}}$ transfers to the internal part. Each transfer involves finding the $\mathcal{M}$ smallest elements among the elements in the buffers of all roots in all collections. First, for each collection, we perform a virtual pull from the semi-sorted roots. We exclude the access cost for the leaves and treat them as semi-sorted. Applying Lemma 5 for each of the $\mathcal{O}(H) = \mathcal{O}\left(\log_\Delta \frac{I}{\mathcal{M}}\right)$ collections leads to a cost of $\mathcal{O}\left(\mathcal{M} \lg \Delta \log_\Delta \frac{I}{\mathcal{M}}\right) = \mathcal{O}\left(\mathcal{M} \lg \frac{I}{\mathcal{M}}\right)$ comparisons and $\mathcal{O}\left(\frac{\mathcal{M}}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)$ I/Os. Then among these $\mathcal{O}\left(\mathcal{M} \log_\Delta \frac{I}{\mathcal{M}}\right)$ elements, we find the $\frac{\mathcal{M}}{2}$ smallest elements by applying Corollary 3. These elements are then transferred to the insert-buffer and removed from their respective roots. This may trigger pulls to restore the invariants, the cost of which we exclude here and is accounted for in Lemma 8. These operations do not asymptotically increase the total cost, which is $\mathcal{O}\left(D \lg \frac{I}{\mathcal{M}}\right)$ comparisons and $\mathcal{O}\left(\frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)$ I/Os. Finally, a transfer to the internal part causes $\mathcal{O}(\mathcal{M})$ comparisons to build the new min-buffer and insert-buffer, which sums up to $\mathcal{O}(D)$ comparisons for all transfers to the internal part. ◀

Proof of Lemma 2. Summarizing the number of comparisons and I/Os performed, the total number of comparisons is

$$\mathcal{O}\Bigg(\underbrace{(I + D \lg \mathcal{M})}_{\text{internal memory}} + \underbrace{\left(I\frac{\lg \Delta}{\Delta} + D \lg \frac{I}{\mathcal{M}}\right)}_{\text{pulls}} + \underbrace{(I + D \lg \Delta)}_{\text{leaves}} + \underbrace{\left(D \lg \frac{I}{\mathcal{M}}\right)}_{\text{transfers}}\Bigg) = \mathcal{O}(I + D \lg I) \ ,$$

and the total number of I/Os is

$$\mathcal{O}\Bigg(\underbrace{\left(\frac{I}{\mathcal{M}} + \frac{DH}{B}\right)}_{\text{pulls}} + \underbrace{\left(\frac{I}{B} + \frac{D}{B}\right)}_{\text{leaves}} + \underbrace{\left(\frac{I}{B} + \frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right)}_{\text{transfers}}\Bigg) = \mathcal{O}\left(\frac{I}{B} + \frac{D}{B} \log_\Delta \frac{I}{\mathcal{M}}\right) \ .$$

Lemma 2 follows from $\mathcal{M} = \Theta(M)$ and $\Delta = \frac{\mathcal{M}}{B}$. ◀

8 Dependence on current priority queue size

In Sections 3–7, I denotes the total number of insertions performed, allowing the current size N to be arbitrarily smaller than I. Since the comparison and I/O bounds of Lemma 2 are dependent on I, the bounds are not a function of the current number of elements N in

the priority queue. To achieve this, we apply the standard technique of *global rebuilding* [28, Chapter 5].

Proof of Theorem 1. Let N_0 denote the size of the priority queue at the last global rebuilding of the priority queue, and let I_0 and D_0 denote the number of insertions and deletions since the last global rebuilding, respectively. We rebuild the priority queue whenever $I_0 + D_0 > \frac{N_0}{2}$. Rebuilding a priority queue containing N elements is performed by scanning over the structure, collecting the N elements into a list, and then repeatedly inserting these into a new empty priority queue using Insert. By Lemma 2, the resulting priority queue is constructed using $\mathcal{O}(N)$ comparisons and $\mathcal{O}\left(\frac{N}{B}\right)$ I/Os (the previous structure can be scanned in $\mathcal{O}\left(\frac{N}{B}\right)$ I/Os, since we argue below that $N = \Theta(N_0)$ and at most $2\frac{N_0+I_0}{\mathcal{M}} = \mathcal{O}\left(\frac{N}{\mathcal{M}}\right)$ buffers have been created in external memory since the last global rebuilding).

Together with Lemma 2, the previous rebuilding and the subsequent I_0 insertions and D_0 deletions, causing a total of $N_0 + I_0$ insertions and D_0 deletions, imply a total comparison cost of $\mathcal{O}(N_0 + I_0 + D_0 \lg(N_0 + I_0)) = \mathcal{O}(N_0 + D_0 \lg N_0)$ and a total I/O cost of $\mathcal{O}\left(\frac{N_0}{B} + \frac{1}{B}\left((N_0 + I_0) + D_0 \log_{M/B} \frac{N_0+I_0}{B}\right)\right) = \mathcal{O}\left(\frac{N_0}{B} + \frac{D_0}{B} \log_{M/B} \frac{N_0}{B}\right)$, since $I_0 \leq \frac{N_0}{2}$. We charge the linear cost to the at least $\frac{N_0}{3}$ priority queue operations between the last two global rebuildings, and the remaining cost to the deletions since the last global rebuilding. While no global rebuilding is triggered, we have $\frac{1}{2}N_0 \leq N \leq \frac{3}{2}N_0$, i.e., we have $N = \Theta(N_0)$ between two global rebuildings. Theorem 1 follows. ◀

9 Conclusion and open problems

The external-memory priority queue presented in this paper supports insertions with asymptotically fewer comparisons and I/Os than previous results. This is particularly beneficial in settings such as event-driven simulations or other incremental computations where execution may terminate before all elements have been extracted from the queue and the cost of insertions dominates. It should be noted that the amortized internal computation time required by our construction is asymptotically the same as the number of comparisons performed (which also holds for the black boxes used by the construction [5, 24, 31]).

The presented data structure is inherently cache-aware. This raises a natural question: does there exist a cache-oblivious priority queue that achieves the same amortized bounds on comparisons and I/Os?

The structure is also highly amortized (e.g., due to global rebuilding, transfers between internal and external memory, recursive pulling, lazy semi-sorted buffers), and a single Insert or DeleteMin operation might require $\Theta(N/B)$ I/Os in the worst-case. Another natural question, is whether it is possible to achieve a solution with worst-case guarantees, e.g., such that a window of B priority queue operations requires worst-case $\mathcal{O}(B \lg N)$ comparisons and $\mathcal{O}\left(\log_{M/B} \frac{N}{B}\right)$ I/Os, while maintaining the improved amortized bounds for Insert.

References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

2 David P. Dobkin andJ. Ian Munro. Optimal time minimal space selection algorithms. *Journal of the ACM*, 28(3):454–461, 1981. doi:10.1145/322261.322264.

3 Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/S00453-003-1021-X.

4 Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal of Computing*, 36(6):1672–1695, 2007. doi:10.1137/S0097539703428324.

5 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

6 Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An experimental study of priority queues in external memory. In Jeffrey Scott Vitter and Christos D. Zaroliagis, editors, *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19-21, 1999, Proceedings*, volume 1668 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1999. doi:10.1007/3-540-48318-7_27.

7 Gerth Stølting Brodal. A survey on priority queues. In Andrej Brodnik, Alejandro López-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, volume 8066 of *Lecture Notes in Computer Science*, pages 150–163. Springer, 2013. doi:10.1007/978-3-642-40273-9_11.

8 Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan J. Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2002. doi:10.1007/3-540-45465-9_37.

9 Gerth Stølting Brodal and Rolf Fagerberg. Funnel heap — A cache oblivious priority queue. In Prosenjit Bose and Pat Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21-23, 2002, Proceedings*, volume 2518 of *Lecture Notes in Computer Science*, pages 219–228. Springer, 2002. doi:10.1007/3-540-36136-7_20.

10 Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In Lawrence L. Larmore and Michel X. Goemans, editors, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 307–315. ACM, 2003. doi:10.1145/780542.780589.

11 Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer, 2004. doi:10.1007/978-3-540-27810-8_41.

12 Gerth Stølting Brodal and Jyrki Katajainen. Worst-case external-memory priority queues. In Stefan Arnborg and Lars Ivansson, editors, *Algorithm Theory - SWAT '98, 6th Scandinavian Workshop on Algorithm Theory, Stockholm, Sweden, July, 8-10, 1998, Proceedings*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 1998. doi:10.1007/BFB0054359.

13 Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7(3):298–319, 1978. doi:10.1137/0207026.

14 Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT 88, 1st Scandinavian Workshop on Algorithm Theory, Halmstad, Sweden, July 5-8, 1988, Proceedings*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1988. doi:10.1007/3-540-19487-8_1.

15 Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings of*

the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain, pages 245–254. ACM, 2004. doi:10.1145/1007912.1007949.

16 James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988. doi:10.1145/50087.50096.

17 Kasper Eenberg, Kasper Green Larsen, and Huacheng Yu. Decreasekeys are expensive for external memory priority queues. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1081–1093. ACM, 2017. doi:10.1145/3055399.3055437.

18 R. Fadel, K. V. Jakobsen, Jyrki Katajainen, and Jukka Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999. doi:10.1016/S0304-3975(99)00006-7.

19 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. doi:10.1145/28869.28874.

20 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 285–298. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814600.

21 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, 2012. doi:10.1145/2071379.2071383.

22 Bernhard Haeupler, Richard Hladík, Václav Rozhon, Robert E. Tarjan, and Jakub Tetek. Universal optimality of Dijkstra via beyond-worst-case heaps. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 2099–2130. IEEE, 2024. doi:10.1109/FOCS61266.2024.00125.

23 Bernhard Haeupler, Richard Hladík, Václav Rozhon, Robert E. Tarjan, and Jakub Tetek. Bidirectional Dijkstra's algorithm is instance-optimal. In Ioana Oriana Bercea and Rasmus Pagh, editors, *2025 Symposium on Simplicity in Algorithms, SOSA 2025, New Orleans, LA, USA, January 13-15, 2025*, pages 202–215. SIAM, 2025. doi:10.1137/1.9781611978315.16.

24 Nicholas J. A. Harvey and Kevin C. Zatloukal. The post-order heap. In *Proceedings Third International Conference on Fun with Algorithms (FUN 2004), Elba, Italy, May 2004*, 2004. URL: http://people.csail.mit.edu/nickh/Publications/PostOrderHeap/FUN04-PostOrderHeap.pdf.

25 John Iacono, Riko Jacob, and Konstantinos Tsakalidis. External memory priority queues with decrease-key and applications to graph algorithms. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 60:1–60:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ESA.2019.60.

26 Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with decreasekeys. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1331–1343. SIAM, 2019. doi:10.1137/1.9781611975482.81.

27 Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996*, pages 169–176. IEEE Computer Society, 1996. doi:10.1109/SPDP.1996.570330.

28 Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. doi:10.1007/BFB0014927.

29 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 263–268. IEEE, 2000. doi:10.1109/IPDPS.2000.845994.

30 Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000. doi:10.1137/S0097539795288246.

31 Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978. doi:10.1145/359460.359478.

32 John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. doi:10.1145/512274.512284.

Chapter 6

Buffered Partially-Persistent External-Memory Search Trees

Buffered Partially-Persistent External-Memory Search Trees

Gerth Stølting Brodal
Aarhus University, Denmark

Casper Moldrup Rysgaard
Aarhus University, Denmark

Rolf Svenning
Aarhus University, Denmark

Abstract

We present an optimal partially-persistent external-memory search tree with amortized I/O bounds matching those achieved by the non-persistent B^ε-tree by Brodal and Fagerberg [SODA 2003]. In a partially-persistent data structure each update creates a new version of the data structure, where all past versions can be queried, but only the current version can be updated. All operations should be efficient with respect to the size N_v of the accessed version v. For any parameter $0 < \varepsilon < 1$, our data structure supports insertions and deletions in amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v\right)$ I/Os, where B is the external-memory block size. It also supports successor and range reporting queries in amortized $\mathcal{O}\left(\frac{1}{\varepsilon} \log_B N_v + K/B\right)$ I/Os, where K is the number of values reported. The space usage of the data structure is linear in the total number of updates. We make the standard and minimal assumption that the internal memory has size $M \geq 2B$. The previous state-of-the-art external-memory partially-persistent search tree by Arge, Danner and Teh [JEA 2003] supports all operations in worst-case $\mathcal{O}(\log_B N_v + K/B)$ I/Os, matching the bounds achieved by the classical B-tree by Bayer and McCreight [Acta Informatica 1972]. Our data structure successfully combines buffering updates with partial persistence. The I/O bounds can also be achieved in the worst-case sense, by slightly modifying our data structure and under the requirement that the memory size $M = \Omega\left(B^{1-\varepsilon} \log_2(\max_v N_v)\right)$. For updates, where the I/O bound is $o(1)$, we assume that the I/Os are performed evenly spread out among the updates (by performing buffer-overflows incrementally). The worst-case result slightly improves the memory requirement over the previous ephemeral external-memory dictionary by Das, Iacono, and Nekrich (ISAAC 2022), who achieved matching worst-case I/O bounds but required $M = \Omega(B \log_B N)$.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases B-tree, buffered updates, partial persistence, external memory

Funding All authors are funded by Independent Research Fund Denmark, grant 9131-00113B.

1 Introduction

Developing data structures for storing a set of values from a totally ordered set subject to insertions, deletions, successor and predecessor queries, and range reporting queries is a fundamental problem in computer science. The classical solution in external-memory is the B-tree by Bayer and McCreight [5] which supports all the operations in worst-case $\mathcal{O}(\log_B N + N/K)$ I/Os, where N is the current size of the set, K is the number of reported values, and B is the external-memory block size. While the B-tree achieves the optimal number of I/Os for queries, for any $0 < \varepsilon < 1$, the B^ε-tree by Brodal and Fagerberg [10] significantly improves update efficiency by attaching buffers to the internal nodes of a B-tree. This design supports updates with amortized $\mathcal{O}\left(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\right)$ I/Os. The $\frac{1}{\varepsilon} B^{1-\varepsilon}$ factor improvement over traditional B-trees is significant when considering typical parameters of, e.g., $\varepsilon = 1/2$ and $B = 1000$ [4] and the B^ε-tree has found important applications in high-performance industry software such as TokuDB [9] and BetrFS [24].

Although the B^ε-tree optimizes update efficiency, it is *ephemeral*, like most dynamic data structures, meaning that each update overwrites the previous version, and only the current version can be queried. In many applications, maintaining access to past versions is beneficial or even essential. A *persistent* data structure supports such accesses, and in their seminal 1989 paper, Driscoll, Sarnak, Sleator, and Tarjan introduced general techniques for making ephemeral data structures persistent [19]. In particular, a *partially-persistent* data structure supports queries in all past versions of the data structure but only the current version can be updated. Multiple authors have adapted these techniques to the external-memory model, developing partially-persistent B-trees that support updates and queries in worst-case $\mathcal{O}(\log_B N_v + K/B)$ I/Os, where N_v is size of the accessed version v, matching the performance of classical B-trees [3, 6, 32].

In this paper, we present the first buffered partially-persistent external-memory search tree that retains the optimal update and query performance of the (ephemeral) B^ε-tree. Our approach combines buffering techniques, which are essential for efficient updates in external memory, with a geometric view of persistence.

1.1 The External-Memory Model

For problems on massive amounts of data that do not fit in internal memory, the standard model of computation is the I/O-model by Aggarwal and Vitter [1]. In this model, all computation occurs in an internal memory of size M, while an infinite external memory is used for storage. Data is transferred between internal and external memory in blocks of B consecutive elements, with each transfer counting as an I/O. The I/O complexity of an algorithm is defined as the total number of I/Os it performs, and the space usage is the maximum number of external-memory blocks used at any given time. The only operation we allow on stored values are comparisons and we follow the standard assumption that the parameters $B \geq 2$ and $M \geq 2B$. Aggarwal and Vitter proved that the optimal bound for sorting in external memory is $\mathit{sort}(N) = \Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os [1]. An algorithm is called *cache oblivious* if it is designed without explicit knowledge of B and M but is still analyzed in the I/O model for arbitrary values of these parameters, assuming an optimal offline cache replacement strategy [21]. Some authors make stronger assumptions on the size of the internal memory, such as the *tall-cache assumption* $M \geq B^{1+\delta}$, for some constant $\delta > 0$. For cache-oblivious algorithms, a tall-cache assumption is necessary to achieve optimal comparison-based external-memory sorting [11].

Being considerate of the I/O-behavior of algorithms can be crucial in practice, as demonstrated by Streaming B-trees [8], the generation of massive graphs for the LFR benchmark [23, 26, 27], and the FlashAttention algorithm used in Transformer models [14].

1.2 Interface of a Partially-Persistent Search Tree

A partially-persistent search tree stores an ordered set of values supporting the below interface (in our examples we use integers, but our data structure works for any totally ordered set). Each version is identified by a unique integer version identifier v, with zero being the initial version and the *current* version denoted by v_c. Further, we let $\mathcal{S}_v$ denote the set of values contained in version v, and N_v the size of $\mathcal{S}_v$. Initially $v_c = 0$ and $\mathcal{S}_{v_c} = \emptyset$. Updates (insertions and deletions) can only be performed on the current set $\mathcal{S}_{v_c}$, and any update advances the current version identifier, i.e., each version of the set $\mathcal{S}_v$ only differs from the previous version $\mathcal{S}_{v-1}$ by at most a single value. Queries can be performed on any version.

INSERT(x) Creates $\mathcal{S}_{v_c+1} = \mathcal{S}_{v_c} \cup \{x\}$, increments v_c, and returns v_c.

Table 1 Overview of results on the I/O complexity of ephemeral and partial persistent search trees. Results marked by "am." hold amortized, and results marked by "rand." are randomized and hold with high probability. All other results hold in worst case. The parameter ε must satisfy $0 < \varepsilon < 1$. All results assume $M = \Omega(B)$, further † assumes $B = \Omega(\log N)$ and $M = \Omega\big(\max\{B \log^{\Theta(1)} N, B^2\}\big)$; ‡ assumes $M = \Omega(B \log_B N)$; and $*$ assumes $M = \Omega\big(B^{1-\varepsilon} \log_2(\max_v N_v)\big)$. For both queries and updates in [7, 15], we include the multiplicative dependency on $\frac{1}{\varepsilon}$ (that can be omitted when treating ε as a constant), allowing, for example, setting $\varepsilon = \frac{1}{\log_2 B}$. All ephemeral results use space linear in N and all partial persistence results use space linear in the total number of updates.

	Range Query	Update
Ephemeral		
Bayer and McCreigh [5]	$\mathcal{O}(\log_B N + K/B)$	$\mathcal{O}(\log_B N)$
Brodal and Fagerberg [10]	$\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N + K/B\big)$ am.	$\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\big)$ am.
Bender, Das, Farach-Colton, Johnson, and Kuszmaul† [7]	$\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N + K/B\big)$	$\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\big)$ rand.
Das, Iacono, and Nekrich‡ [15]	$\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N + K/B\big)$	$\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N\big)$
Partial Persistent		
Becker, Gschwind, Ohler, Seeger, and Widmayer [6]	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v)$
Varman and Verma [32]		
Arge, Danner, and Teh [3]		
This paper (Theorem 1)	$\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N_v + K/B\big)$ am.	$\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v\big)$ am.
*This paper (Theorem 2)**	$\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N_v + K/B\big)$	$\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B N_v\big)$
This paper (Theorem 3)	$\mathcal{O}\big(\frac{1}{\varepsilon} \log_B N_v + \gamma + K/B\big)$	$\mathcal{O}\big(\frac{1}{B^{1-\varepsilon}} \big(\frac{1}{\varepsilon} \log_B N_v + \gamma\big)\big)$
	$\gamma = sort\big(B^{1-\varepsilon} \log_2 N_v\big) = \frac{\log_2 N_v}{B^\varepsilon} \log_{M/B} \frac{\log_2 N_v}{B^\varepsilon}$	

DELETE(x) Creates $\mathcal{S}_{v_c+1} = \mathcal{S}_{v_c} \setminus \{x\}$, increments v_c, and returns v_c.
RANGE(v, x, y) Reports all values in $\mathcal{S}_v \cap [x, y]$ in increasing order.
SEARCH(v, x) Returns the successor of x in $\mathcal{S}_v$, i.e., $\min\{y \in \mathcal{S}_v \mid x \leq y\}$.

1.3 Previous Work

In internal memory, the *fat node* and *node copying* techniques can make any ephemeral linked data structure partially-persistent with constant overhead in both time and space, as long as the in-degree of each node in the ephemeral structure is constant [19]. Becker, Gschwind, Ohler, Seeger, and Widmayer [6] and Varman and Verma [32] adapted these techniques to B-trees in external-memory. An elegant application of partial persistence appears in the design of linear space planar point location data structures [31]. In this setting, the underlying set consists of segments which are partially ordered (only a pair of segments intersected by a vertical line can be compared). To adapt this approach to the external-memory setting, Arge, Danner, and Teh strengthened the partially-persistent B-tree to require only a total order on values alive at any given version, leading to a static external-memory point-location structure [3].

A different approach to persistence is to interpret it geometrically (Figure 1), modeling it as a data structure problem on a dynamic set of vertical (or horizontal) segments. Kolovson and Stonebraker explored this perspective [25], though their reliance on R-trees led to poor

performance guarantees [22]. More recently, Brodal, Rysgaard, and Svenning [12] leveraged this geometric approach to develop *fully persistent* B-trees, which allow both queries and modifications to all past versions in $\mathcal{O}(\log_B N_v)$ I/Os. In a fully persistent data structure, updating a version corresponds to cloning it and then applying the modification to the newly cloned version, ensuring that existing versions remain unaffected. Such behavior contrasts with *retroactive data structures* [16], where updates recursively propagate to cloned versions.

Concurrently with the work on persistent data structures in external-memory, there were significant improvements to external-memory data structures by leveraging buffering techniques to always process multiple updates and/or queries together. These include the Buffer Tree by Arge [2] which can form the basis for external-memory sorting, priority queues and batched dynamic algorithms [20] in amortized $\mathcal{O}\Big(\frac{1}{B}\log_{M/B}\frac{N}{B}\Big)$ I/Os per operation. For a batched operation the answer might not be immediately returned, which is often sufficient, e.g., in many geometric plane-sweep algorithms where only the end result matters. For standard (non-batched) data structures, a line of work has investigated the update-query trade-off, beginning with the Buffered Repository Tree [13] performing updates in amortized $\mathcal{O}\big(\frac{1}{B}\log_B N\big)$ I/Os and queries in $\mathcal{O}(\log_2 N)$ I/Os. This was later generalized by the B^ε-tree [10] which for $\varepsilon \approx 0$ corresponds to the Buffered Repository Tree and for $\varepsilon \approx 1$ to the standard B-tree. The amortized performance of the B^ε-tree was improved to high-probability [7] and worst-case [15] I/O bounds using stronger assumptions on the size of B and M (see Table 1).

1.4 Contribution

Combining the two lines of research on persistence and buffered data structures has remained an open challenge for the past 20 years, likely due to their seemingly conflicting principles. Persistence requires maintaining access to past versions without affecting their structure, while buffers essentially hold updates to past versions before applying them. Our work demonstrates that that these two ideas can be effectively unified by developing partially-persistent external-memory search trees that achieve bounds matching those of ephemeral B^ε-trees.

▶ **Theorem 1.** *Given any parameter $0 < \varepsilon < 1$ and $M \geq 2B$, there exist partially-persistent external-memory search trees over any totally ordered set, that support* Insert *and* Delete *in amortized $\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N_v\big)$ I/Os,* Search *in amortized $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v\big)$ I/Os, and* Range *in amortized $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v + K/B\big)$ I/Os. Here N_v denotes the number of values contained in version v, and K the number of values reported by* Range. *The space usage is linear in the total number of updates.*

The query Search can trivially also answer a member query "$x \in \mathcal{S}_v$?" by checking if Search(v, x) returns x. Our data structure can further also support predecessor queries instead of successor queries, as well as strict predecessor and successor queries, i.e., the returned value should be strictly smaller or larger than the query value x. The structure can also handle the case when $\mathcal{S}_0 \neq \emptyset$, where the initial structure can be constructed using $\mathcal{O}(1 + |\mathcal{S}_0|/B)$ I/Os (essentially this is Section 2.6, where a structure is constructed for a given set). Our data structure is stated as maintaining a set of values, but it can easily be extended to support dictionaries storing key-value pairs (each segment in Figure 1 and Figure 2 now stores a key-value pair, where the first axis now are keys; changing the value for key x at version v starts a new vertical segment at (x, v) with the new value).

In Section 3 we describe how to convert the amortized I/O bounds of Theorem 1 to worst-case bounds under the assumption that $M = \Omega\big(B^{1-\varepsilon}\log_2(\max_v N_v)\big)$ (Theorem 2),

a weaker or equal assumption on the memory size than used in [7] and [15] for high-probability and worst-case bounds for B^ε-trees, respectively. Under the weakest assumption that $M \geq 2B$, we achieve the worst-case bounds in Theorem 3 with an additional term of at most $\mathcal{O}\big(sort\big(B^{1-\varepsilon}\log_2 N_v\big)\big)$ I/Os, where $B^{1-\varepsilon}\log_2 N_v$ is an upper bound on the number of buffered updates on a root-to-leaf path that should be flushed to the leaf, and $sort(N) = \Theta\Big(\frac{N}{B}\log_{M/B}\frac{N}{B}\Big)$ denotes the number of I/Os to sort N values [1]. For updates, where the I/O bound is $o(1)$, we assume that the I/Os are performed evenly spread out among the updates.

▶ **Theorem 2.** *Given any parameter $0 < \varepsilon < 1$ and $M = \Omega\big(B^{1-\varepsilon}\log_2(\max_v N_v)\big)$, there exist partially-persistent external-memory search trees over any totally ordered set, that support* Insert *and* Delete *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N_v\big)$ I/Os,* Search *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v\big)$ I/Os, and* Range *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v + K/B\big)$ I/Os. Here N_v denotes the number of values contained in version v, and K the number of values reported by* Range*. The space usage is linear in the total number of updates.*

▶ **Theorem 3.** *Given any parameter $0 < \varepsilon < 1$ and $M \geq 2B$, there exist partially-persistent external-memory search trees over any totally ordered set, that support* Insert *and* Delete *in worst-case $\mathcal{O}\big(\frac{1}{B^{1-\varepsilon}}\big(\frac{1}{\varepsilon}\log_B N_v + \gamma\big)\big)$ I/Os,* Search *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v + \gamma\big)$ I/Os, and* Range *in worst-case $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v + \gamma + K/B\big)$ I/Os, where $\gamma = sort\big(B^{1-\varepsilon}\log_2 N_v\big)$. Here N_v denotes the number of values contained in version v, and K the number of values reported by* Range*. The space usage is linear in the total number of updates.*

Note that, for example, when $N_v = 2^{\mathcal{O}(B^\varepsilon)}$ then $B^{1-\varepsilon}\log_2 N_v = \mathcal{O}(B)$ and $\gamma = \mathcal{O}(1)$ and the I/O bounds of Theorem 3 match those of Theorem 2, with only the assumption $M \geq 2B$. This observation can be further strengthened, as when $\gamma = \mathcal{O}\big(\frac{1}{\varepsilon}\log_B N_v\big)$ the bounds I/O match similarly, which holds when $N_v = 2^{B^\varepsilon\left(\frac{M}{B}\right)^{\mathcal{O}\left(\frac{B^\varepsilon}{\varepsilon\log_2 B}\right)}}$.

Outline of Data Structure Previous work on partially-persistent search trees in external memory directly adapted the general pointer-based transformations for persistence [19]. In contrast, our approach embraces the geometric interpretation of partial persistence (see Figure 1) similar to that of [12], where the state of the data structure is embedded in a two-dimensional plane with values on the first axis and versions on the second axis. Under this interpretation, each update corresponds to the start or end of a vertical segment in the plane. Since partial persistence updates are applied to the current version, it always affects the top of the plane. Successor and predecessor queries correspond to horizontal ray shooting to the right and left, respectively, and range queries to reporting the intersections between a horizontal query segment among vertical segments.

To efficiently update and query the geometric view, we partition the plane into rectangles, each containing $\Theta\big(\frac{1}{\varepsilon}B\log_B \overline{N}\big)$ vertical segments in lexicographic order. For now we assume that all versions have size $\Theta\big(\overline{N}\big)$, for a fixed $\overline{N}$ (this assumption is lifted using global rebuilding, see Section 2.6). In the geometric persistent view, a vertical segment crossing multiple rectangles is split into multiple smaller segments, one for each rectangle, and each smaller segment is inserted into one rectangle.

At a high level, our data structure is divided into two parts. The top part consists of all the open rectangles containing the current set $\mathcal{S}_{v_c}$, which may still be updated. The entry point of this data structure is a B^ε-tree T on the value axis to facilitate buffered updates and to find the relevant rectangle(s) for updates and queries. Since updates are buffered, the geometric view stored in the rectangles may be incomplete, since buffered updates (segment

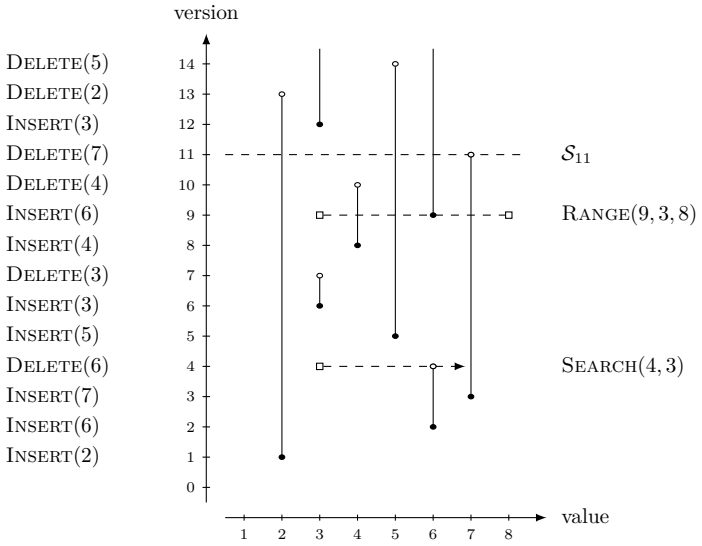


Figure 1 (Left) A list of updates performed on an initially empty set and (Right) the geometric interpretation of the updates. Vertical lines illustrate the interval of versions containing a value. Note that the value 4 is contained in versions $[8, 10[$, i.e., versions 8 and 9, whereas the value 3 is contained in version 6 and versions $[12, \infty[$. The topmost dashed line shows that version 11 of the set is $\mathcal{S}_{11} = \{2, 5, 6\}$, the dashed line segment at version 9 shows that the result of the query Range$(9, 3, 8)$ is $\{4, 5, 6, 7\}$, and the bottommost dashed arrow shows that the result of the successor search Search$(4, 3)$ is 7.

endpoints) will first be added to the rectangle when buffers are flushed. The bottom part consists of all the finalized rectangles, i.e., rectangles which can be queried but not updated. The entry point to the bottom part is a data structure P to find the relevant rectangle(s) for a query. This corresponds exactly to a point location problem and we implement P as an external-memory adaption of the classical planar point location solution using partial persistence [3, 31], more specifically a B-tree with path copying during updates.

2 The Buffered Persistent Data Structure

In this section, we describe our partially-persistent B^ε-tree structure. Versions are identified be the integers $0, 1, 2, \ldots$, where v_c denotes the identifier of the current version. We let $\mathcal{S}_v$ denote the set at version v, where values are from some totally ordered set. The initial set $\mathcal{S}_0 = \emptyset$, and $\mathcal{S}_{v+1} = \mathcal{S}_v \cup \{x\}$ if the $v+1$'th update is Insert(x), and $\mathcal{S}_{v+1} = \mathcal{S}_v \setminus \{x\}$ if the $v+1$'th update is Delete(x). Note that $\mathcal{S}_{v+1} = \mathcal{S}_v$ if the $(v+1)$'th update inserts a value already in $\mathcal{S}_v$ or is deleting a value not in $\mathcal{S}_v$.

2.1 Geometric Interpretation of Partial Persistence

The problem has a natural geometric interpretation in a two dimensional space, with the first dimension representing the values and the second dimension representing the versions, see Figure 1. On this two dimensional plane, a value x existing in versions $[v, w[$, can be represented by the vertical line segment $\{x\} \times [v, w[$, i.e., x is inserted in version v and deleted in version w. If $x \in \mathcal{S}_{v_c}$, then $w = +\infty$ (x has not been deleted yet).

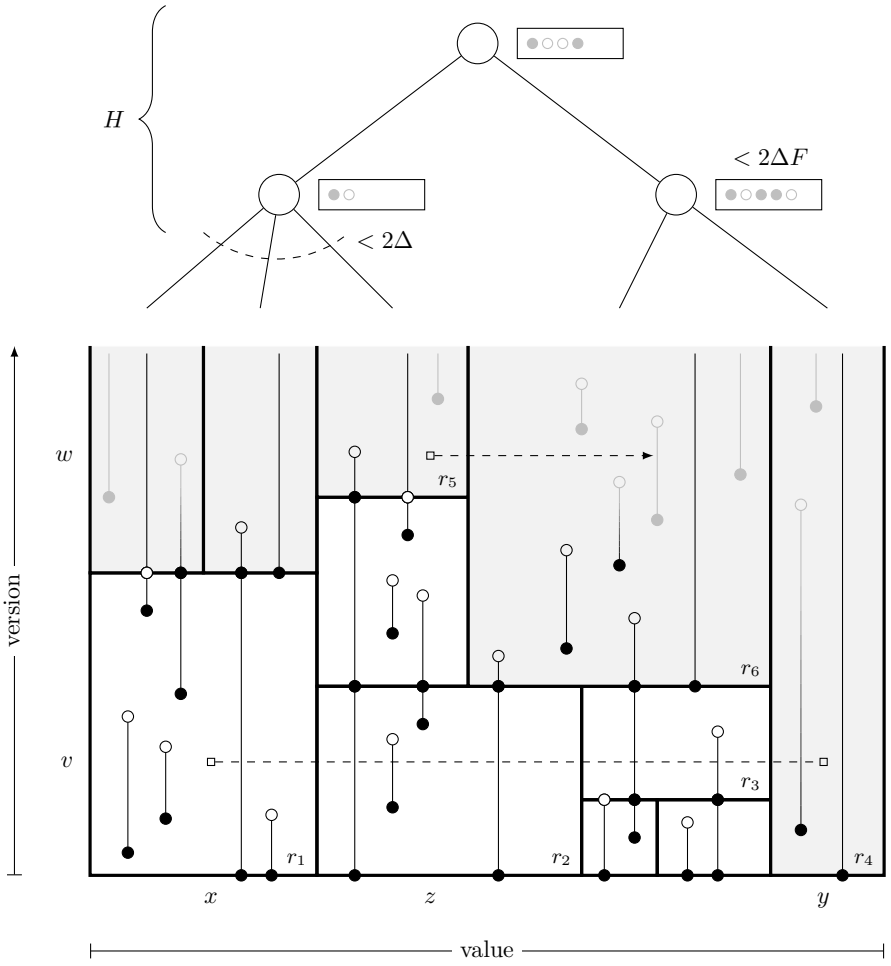


Figure 2 (Top) A B^ε-tree of the open rectangles and (Bottom) the geometric interpretation of the updates, split into multiple rectangles, where gray rectangles are open rectangles. The black endpoints represent updates present in the rectangle and the gray endpoints represent buffered updates present in the buffers at the internal nodes of the B^ε-tree. A query $\textsc{Range}(v, x, y)$ is represented as the dashed line between two square endpoints, spanning rectangles r_1, r_2, r_3, and r_4, and a successor query $\textsc{Search}(w, z)$ is represented as the dashed arrow from a square endpoint, spanning rectangles r_5 and r_6. Black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

2.2 Partitioning the Plane into Rectangles

We consider the sequence of versions partitioned into intervals $[v_0, v_1[, \ldots, [v_{k-1}, v_k[, [v_k, \infty[$, for some versions $0 = v_0 < v_1 < \cdots < v_k \leq v_c$. In Section 2.6 we show how to maintain the version intervals. In the following we consider the interval $[\overline{v}, \infty[$ containing the current version v_c of the set. Let $\overline{N} = |\mathcal{S}_{\overline{v}}|$. We allow up to $c \cdot \overline{N}$ partial persistent updates during this interval of versions for a constant $0 < c < 1$, i.e., all versions v, $\overline{v} \leq v \leq v_c$, satisfy $(1-c) \cdot \overline{N} \leq |\mathcal{S}_v| \leq (1+c) \cdot \overline{N}$.

Our data structure is built around four central parameters:

$$\Delta = \lceil B^\varepsilon \rceil \qquad H = 1 + \lceil \log_\Delta \overline{N} \rceil \qquad F = \lceil B^{1-\varepsilon} \rceil \qquad R = H \cdot 2\Delta \cdot F$$

The basic idea is to have a B^ε-tree T of degree at most 2Δ (and degree at least Δ, if only

insertions can be performed, and degree at least 1 if deletions are allowed), where leaves (open rectangles) store between $4R$ and $10R$ updates (see Section 2.4) and all leaves are at the same layer. In Section 2.5 we prove that H is an upper bound on the height of T (number of nodes on a root-to-leaf path, excluding the leaves). Each internal node of T will have a buffer of at most $2\Delta F = \Theta(B)$ updates yet to be applied to the leaves of the subtree rooted at the node. Note that R is an upper bound on the total number of buffered updates along a root-to-leaf path in T. The essential property of the parameters is that $R/B = \Theta(H) = \Theta\big(\frac{1}{\varepsilon}\log_B \overline{N}\big)$.

The geometric plane defined in Section 2.1 is partitioned into a number of axis aligned rectangles $[x, y[\times[v, w[$, such that the number of updates in each rectangle is $\Theta(R)$. For each rectangle we store a list of the updates in the rectangle in lexicographical order by first the value and secondly the version of that update. Note that this groups equal values consecutively in the list. To allow efficiently locating a rectangle for a given version and value, we store a list indexed by version identifier, where we for each version v store a pointer to the root of a B-tree P_v over the rectangles left-to-right containing $\mathcal{S}_v$ (see Section 2.4). Further, we require that each rectangle contains $\Omega(R)$ values which are present in all versions the rectangle spans. We denote such a value as *spanning*. If $\overline{N} = \mathcal{O}(R)$, all updates are stored in a single list.

New updates are buffered, to achieve I/O efficient update bounds. The topmost rectangles, which cover the current version v_c, are all *open*, with all other rectangles being *closed*. Crucially, new updates are always performed in the current version. We maintain the invariant that for a buffered update, i.e., an update that has not yet been flushed to the corresponding rectangle, the corresponding rectangle must be open.

For the open rectangles, we store a B^ε-tree T, such that recent updates to the open rectangles are buffered. We let the maximum degree of an internal node in T be 2Δ. Each internal node of T contains a buffer of up to $2\Delta F$ updates, sorted lexicographically by value and version. Additionally, each update stores if the update is an insertion or deletion. Consider a full buffer, i.e., it contains at least $2\Delta F$ updates, where each update should be *flushed* to one of the at most 2Δ children. Then, there must exist a subset of size at least F updates, which should be flushed to the same child.

The setup is illustrated on Figure 2. The vertical black and gray lines represent the version intervals containing a value. The black lines represent updates present in the list of updates contained in that rectangle, while the gray lines and endpoints represent updates contained in buffers of T, which are illustrated at the top of the figure.

2.3 Handling Queries and Updates

When performing $\textsc{Search}(v, x)$, first the rectangle r covering point (x, v) in the plane must be found. By using the B-tree P_v associated with version v, r can be found using $\mathcal{O}(H)$ I/Os. If r is closed, then all updates inside r are contained in the sorted list of updates stored in r, and these can be scanned in $\mathcal{O}(R/B)$ I/Os. If r is open, then the result of the successor query may be affected by buffered updates, which are not stored in r. The rectangle r must therefore be *actualized*, by merging all updates in buffers on the path from the root to r with the updates in r. The details of this operation is described in Section 2.4, where the actualize operation is shown to have an amortized $\mathcal{O}(H)$ I/Os. After r is actualized, the query continues by scanning the updates of r. If the result of the successor query is not contained in the rectangle r, then by the spanning requirement, the result of the query must be in the neighboring rectangle to the right, that similarly is actualized if it is open. In total, the operation spends amortized $\mathcal{O}(H + R/B) = \mathcal{O}\big(\frac{1}{\varepsilon}\log_B \overline{N}\big)$ I/Os. Note that the operation

easily can be modified to support member, predecessor, and strict predecessor or successor queries.

A RANGE(v, x, y) query is performed very similar to a SEARCH query. The query may however touch more than two rectangles. Note that for the at most two rectangles containing the endpoints of the query we do not necessarily report all values they contain at version v. These rectangles can be found using amortized $\mathcal{O}(H + R/B)$ I/Os by the argument above. For each intermediate rectangle accessed (and possibly actualized if it is open), then by the spanning requirement, a constant fraction of the updates scanned result in reported values. Since accessing a rectangle takes amortized $\mathcal{O}(H + R/B)$ I/Os, and each rectangle contains $\Theta(R) = \Theta(B \cdot H)$ values at version v, then amortized $\mathcal{O}(1/B)$ I/Os are spent for each value reported for an intermediate rectangle. In total, a RANGE query reporting K values takes amortized $\mathcal{O}(H + R/B + K/B) = \mathcal{O}\big(\frac{1}{\varepsilon} \log_B \overline{N} + K/B\big)$ I/Os.

Each update, either an INSERT or DELETE, is applied to the current version v_c of the set. The B^ε-tree T contains all buffered updates to the open rectangles, which cover the current version v_c. For an update operation, a tuple with the update and v_c is added to the root buffer of T, which is stored in internal memory. In Section 2.4 it is shown that adding the update to the root buffer and handling possible *buffer overflows* takes amortized $\mathcal{O}(H/F) = \mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}} \log_B \overline{N}\big)$ I/Os.

2.4 Flushing Buffers

To argue about the amortized cost of flushing the content of buffers down the tree T, we let the potential of each buffered update be $1/F$ multiplied by the height of the buffer the update is stored in, with the root buffer being at the largest height. One unit of released potential can cover $\mathcal{O}(1)$ I/Os. When adding an update to the tree, the root buffer is always stored in internal memory, and therefore no I/Os are needed to access it. However, the potential is increased by at most $1/F \cdot H$, and the operation therefore uses amortized $\mathcal{O}(H/F)$ I/Os.

Buffer Overflows. Each buffer at an internal node of T contains at most $2\Delta F$ updates. If a buffer contains more than $2\Delta F$ updates, then a *buffer overflow* is performed. Since each node of T has at most $2\Delta - 1$ children, at least F updates from the buffer must be to the same child. These updates can be moved together to the buffer of that child.

A buffer overflow can happen in two cases. Either when an update is placed into the root buffer as the result of an update operation, or when updates are placed into a buffer because the parent buffer is overflowing. Moving exactly F updates out of a buffer, is always sufficient to make an overflowing buffer non-overflowing again. A buffer overflow therefore only moves down a single path of T.

An overflowing buffer can be stored in $\mathcal{O}(1)$ blocks, since $2\Delta F + F = \mathcal{O}(B)$, and therefore the F updates to remove can be found in $\mathcal{O}(1)$ I/Os. If the overflowing updates are moved to a child buffer, these can be inserted via a merge in $\mathcal{O}(1)$ I/Os. As F updates are moved one layer down the tree, then the potential decreases by 1, which is enough to cover the $\mathcal{O}(1)$ I/O cost of the overflow operation.

If the child is not an internal node of T, but an open rectangle, then merging the F overflowing updates into the list of updates in the rectangle takes $\mathcal{O}(R/B) = \mathcal{O}(H)$ I/Os. As buffer overflows only move down a single path of the tree, then $\Omega(H)$ overflows must have occurred before the overflow reaches the open rectangle. Merging the overflow into the list of updates in the open rectangle therefore does not increase the asymptotic amortized number of I/Os performed.

Actualizing. When *actualizing* an open rectangle, all buffered updates to that open rectangle must be moved into the rectangle. Note that all relevant updates are in the buffers on the root-to-leaf path in T to the open rectangle. Each of these buffers contains at most $2\Delta F = \mathcal{O}(B)$ buffered updates. The total number of buffered updates on the path is at most $H \cdot 2\Delta \cdot F = R$. For each node on the path from the root down to the rectangle the following is done. Let U be the updates on the path from the layers above in sorted order. Initially U is empty. To extend U for each layer top-down, U is merged with the relevant updates of the next buffer. This requires $\mathcal{O}(1 + |U|/B)$ I/Os, by scanning U and the buffer. Since the U updates are moved one layer down, they release potential $|U|/F \geq |U|/B$, that can cover $\Theta(|U|/B)$ I/Os, i.e., the amortized cost for actualizing one level of the tree is $\mathcal{O}(1)$ I/Os. As there are $\mathcal{O}(H)$ layers of the tree, the at most R relevant updates can be found in sorted order in amortized $\mathcal{O}(H)$ I/Os. They can then be merged with the updates in the open rectangle in $\mathcal{O}(R/B)$ I/Os. In total, the actualize operation requires amortized $\mathcal{O}(H + R/B) = \mathcal{O}(H)$ I/Os.

Finalizing. Each open rectangle is allowed to receive between R and $2R$ updates before it is *finalized*, converting it into a closed rectangle. When finalizing an open rectangle, all buffered updates to the rectangle are removed from T and merged with the rectangle, to ensure that all buffered updates in T are only to open rectangles. We will now argue that open rectangles receive at most $2R$ updates in total by finalizing the rectangle as soon as R updates have been added to it. A finalize operation can be triggered from an actualize operation or from a buffer overflow. Note that in both cases, the number of updates in the rectangle before the operation is at most $R - 1$.

An actualize operation may add at most R buffered updates to a rectangle, i.e., at most $2R - 1$ total updates are placed in a finalized rectangle. If the rectangle receives an update from a buffer overflow, then the overflow must have been triggered by an update in the root buffer. Buffered updates to add to the rectangle can only be the R updates in buffers on the path, and the new update, which in total is at most $(R - 1) + R + 1 = 2R$ updates to add to the open rectangle. Thus, by finalizing a rectangle as soon as it receives at least R updates, it will contain between R and $2R$ updates.

Spanning Requirement. We require that the first version of a rectangle contains $[4R, 8R[$ values. When finalizing the rectangle, $[R, 2R]$ updates have been performed and therefore at least $2R$ of the initial values are still present, that is, span all versions of the rectangle. This ensures that the $\Omega(R)$ spanning values requirement is met.

When finalizing a rectangle, it holds that $[2R, 10R[$ values are contained in the rectangle at version v_c. New open rectangles must be created to span the value range of the closed rectangle, where the values present at version v_c are contained. If the count is in $[4R, 8R[$, then a single rectangle suffices. If $[8R, 10R[$ values are present, then the range is *split* in two rectangles at the median value, both with $[4R, 5R[$ values, and T must be updated as described below. Otherwise, version v_c of the rectangle contains $[2R, 4R[$ values. A sibling rectangle is finalized, to allow for a *merge* of the rectangles to occur. Note that the early finalizing of the sibling preserves the $\Omega(R)$ spanning values requirement of the sibling. The combined present values is then $[4R, 14R[$. A split may need to be performed, i.e., the result is one or two new open rectangles.

Updating the B^ε-Tree T. After finalizing open rectangles, the B^ε-tree must be updated accordingly. If an open rectangle was split, then a new child is added to the parent node

in T of the updated rectangle. If this increases the degree of the node to 2Δ, it is split by distributing its children into two new nodes, each with degree Δ. Its buffer is also partitioned so that each buffered update is placed in the buffer containing its relevant child. Splitting the node further introduces a new child in the parent of the node. Note that this may cascade up the tree, but only on the path towards the root. If a merge of the rectangle occurred, then a child is deleted from the parent. When merging rectangles, the merged rectangles must be siblings in the tree. The rectangle is merged with the left or right neighbor rectangle, which has a closest lowest common ancestor with the rectangle in T, to ensures that the value range of existing nodes only increase. This may cause the degree of nodes to be below Δ. Notably, we do not merge internal nodes of T, as this could create a large buffer that requires multiple flushes in different directions, known as *flushing cascades* [7]. We instead allow nodes to have a degree down to 1, where deleting the last child results in deleting the path of consecutive degree-one from the child towards the root. As we show in Section 2.5, this does not affect the asymptotic height of the tree.

When finalizing a rectangle, only the path from the finalized rectangle to the root may be affected. We therefore create the new tree T via *path copying*, which preserves the old T. The buffers of the copied nodes are moved, such that all buffers are present in the tree for the current version. We maintain a list indexed by version identifier, that for any version v stores a pointer to the root of T for version v, i.e., the required tree P_v.

Updating the rectangles and the B^ε-tree T upon finalizing therefore requires scanning $\mathcal{O}(R)$ values and traversing a constant number of paths of length $\mathcal{O}(H)$ in T, which takes $\mathcal{O}(R/B + H) = \mathcal{O}(R/B)$ I/Os. As $\Omega(R)$ updates must be applied to a rectangle before finalizing it, this does not increase the asymptotic amortized cost of an update operation. Queries may also finalize rectangles, but already require amortized $\mathcal{O}(H)$ I/Os, causing no asymptotic query overhead.

Space Usage. When finalizing a rectangle, $\Omega(R)$ updates must have occurred in that rectangle. New rectangles are then created, which in total copies $\mathcal{O}(R)$ updates, and one path of the tree is copied. As the height of the tree is at most $H = \mathcal{O}(R/B)$, and the updates of the rectangles are stored in lists, the newly allocated space is $\mathcal{O}(R/B)$, which can be amortized over the $\Omega(R)$ updates required for the finalization to happen. In addition to the updates, initially $\overline{N}$ values are stored across $\mathcal{O}\big(\overline{N}/R\big)$ rectangles in lists, and a balanced initial B^ε-tree is built on these initial rectangles, causing an initial space of $\mathcal{O}\big(\overline{N}/B\big)$ blocks. As the structure allows for at most $c \cdot \overline{N}$ updates, the space usage is therefore in total $\mathcal{O}\big(\overline{N}/B\big)$ blocks.

2.5 Bounding the Tree Height

In this section we show that H is an upper bound on the height of the B^ε-tree T.

We define the *weight* w_i of a node at height i in T to be the number of updates on values in the value range of the node. The rectangles are at height 0 of the tree, with the nodes of the tree starting at height 1. The updates are both the $\overline{N}$ initial values as well as the up to $c \cdot \overline{N}$ additional updates. It holds that the weight of a node is the sum of the weights of its children. By induction on the number of updates we show that $w_i \geq B\Delta^i$ for all nodes at all heights, except for the root. First note that the inequality holds for $i = 0$, as any rectangle contains at least $R \geq B$ updates when it was created. Initially, $\overline{N}$ updates are distributed into at most $\overline{N}/R$ rectangles, where the number of updates in each rectangle is at least $R \geq B$. Each internal node initially has degree $[\Delta, 2\Delta[$, except for the root that has degree $[2, 2\Delta[$. By induction on the tree height i, it holds that the initial tree

satisfies $w_i \geq B\Delta^i$, except for the root. Each update affects some path of the tree. If the tree is not updated, then the weights of the nodes on the path can only grow, and therefore the inequality holds. If rectangles are merged, then one rectangle disappears together with all the ancestors having only this single rectangle as a leaf. The other rectangle and its ancestors up to the least common ancestor of the two merged leaves get their value ranges expanded. It follows that the surviving nodes of a merge only can have their value range increase, and therefore the inequality holds. If a split occurs in any node at height i, then the degree of the node before the split is 2Δ. The node is split in two nodes at height i, each with Δ children. The weight of each of the two nodes is therefore at least $\Delta \cdot w_{i-1} \geq \Delta \cdot B\Delta^{i-1} = B\Delta^i$. It therefore holds that $w_i \geq B\Delta^i$.

Since the number of updates is at most $(1+c)\cdot\overline{N}$, we have $B\Delta^i \leq (1+c)\cdot\overline{N}$ for all nodes at height i, except for the root. Since by definition $2 \leq \Delta \leq B$ and $c < 1$, we have $\Delta^{i+1} \leq 2\overline{N}$, i.e., $i \leq \log_\Delta\left(2\overline{N}\right) - 1 \leq \log_\Delta \overline{N}$. The height of T is then at most the largest value of i satisfying this inequality, plus one for the root, i.e., the height of T is at most $1 + \log_\Delta \overline{N} \leq 1 + \lceil \log_\Delta \overline{N} \rceil = H$.

2.6 Global Rebuilding

The data structure above allows for an initial set of $\overline{N}$ values to receive up to $c\cdot\overline{N}$ persistent updates, for a constant $0 < c < 1$. For any version v, we have $(1-c)\cdot\overline{N} \leq N_v \leq (1+c)\cdot\overline{N}$, i.e., $N_v = \Theta\left(\overline{N}\right)$. Therefore, the asymptotic costs of all operations also hold with $\overline{N}$ replaced by N_v.

To allow for more than $c\cdot\overline{N}$ updates, we create multiple copies of the data structure above using global rebuilding [28, 29]. Whenever the current data structure reaches $c\cdot\overline{N}$ updates, a new data structure is created with initial set $\mathcal{S}_{v_c}$ and $\overline{N}_{\text{new}} = |\mathcal{S}_{v_c}|$ (and new H and R parameters), with a new set of rectangles and a new B^ε-tree T, where all buffers are empty. We compute $\mathcal{S}_{v_c}$ by performing $\textsc{Range}(v_c, -\infty, \infty)$ in amortized $\mathcal{O}\left(\frac{1}{\varepsilon}\log_B \overline{N} + K/B\right) = \mathcal{O}\left(\overline{N}/B\right)$ I/Os. The new data structure can be build using $\mathcal{O}\left(\overline{N}/B\right)$ I/Os by a single scan of the sorted list containing $\mathcal{S}_{v_c}$.

In the old data structure $c\cdot\overline{N}$ updates have been performed before this rebuild is performed. By amortizing the rebuild cost over these updates, the amortized cost of each update is increased by $\mathcal{O}(1/B)$ I/Os, i.e., the asymptotic amortized cost of an update is not increased. As the space usage of the new data structure is $\mathcal{O}\left(\overline{N}/B\right)$ blocks, a similar argument can be used to amortize the space usage over the updates, maintaining a linear space usage in the total number of updates. This concludes the proof of Theorem 1.

3 Worst-Case Bounds

In this section, we describe how to achieve worst-case I/O guarantees instead of amortized under progressively weaker assumptions on the internal memory size M. Previous approaches to improving the amortized performance of ephemeral B^ε-trees, both in the randomized [7] and worst-case [15] setting, assumed at least that $M = \Omega\left(\frac{1}{\varepsilon}B\log_B \overline{N}\right) = \Omega(HB)$, which allows all buffers on a path to be sorted in internal memory, i.e., $\mathcal{O}(sort(HB)) = \mathcal{O}(H)$ I/Os. First, in Section 3.1, we show that if $M = \Omega(HB)$, our persistent structure can be deamortized without asymptotic overhead. Then, in Section 3.2, we relax the assumption to $M = \Omega\left(B^{1-\varepsilon}\log_2 \overline{N}\right)$ using the *subtracting game* studied by Dietz and Raman [17]. This represents a factor $B^\varepsilon/\log_2 B$ improvement on the assumption for the size of the internal memory. Finally, in Section 3.3, we show worst-case results when only assuming $M \geq 2B$,

which introduces a small additive overhead on all operations. We employ *the zeroing* game by Dietz and Sleator [18, Theorem 5] to avoid a multiplicative overhead for RANGE queries.

3.1 Large Internal Memory Assumption

We first consider the case when $M = \Omega(HB)$. When actualizing a rectangle (see Section 2.3), the buffers at the $\mathcal{O}(H)$ nodes along the root-to-leaf path, with a total size of $\mathcal{O}(HB)$, are merged to produce a sorted list of updates to apply to the rectangle. As shown in Section 2.4, this can be done in amortized $\mathcal{O}(H)$ I/Os, by merging the buffers top-down. In the worst case, this requires $\mathcal{O}(H^2)$ I/Os. By instead merging the buffers using an external memory sorting algorithm, the worst-case number of I/Os can be improved to $\mathcal{O}(sort(HB))$. Previous approaches to improving the amortized performance of B^ε-trees in the randomized [7] and worst-case [15] settings both assumed at least that $M = \Omega(HB)$, in which case the sorting term trivially disappears by performing the sorting internally after reading the H buffers into internal memory. The remaining challenge was handling flushing cascades, which occur when merging internal nodes of T results in large buffers requiring many flushes in different directions. For our structure, we avoid this issue by never merging internal nodes, and instead maintain the height of T using global rebuilding. For the remainder of this section, we assume a large internal memory of size $M = \Omega(HB)$ and describe how to achieve worst-case guarantees by incrementally performing amortized work.

Queries. Finding and actualizing a relevant rectangle for a query takes $\mathcal{O}(H + sort(HB)) = \mathcal{O}(H)$ I/Os when $M = \Omega(HB)$. The worst case for a SEARCH and RANGE query is therefore $\mathcal{O}(H)$ and $\mathcal{O}((1 + K/R)\,H) = \mathcal{O}(H + K/B)$ I/Os, respectively. Note that for a RANGE query, for each rectangle that intersects the query, except for the leftmost and rightmost ones, $\Omega(R)$ values are reported due to the $\Omega(R)$ spanning values in each rectangle.

Updates. When performing an update, it may be the $\lceil c \cdot \overline{N} \rceil$'th update, which triggers a global rebuild of the structure based on a new $\overline{N}$, which uses $\mathcal{O}(\overline{N}/B)$ I/Os, as described in Section 2.6. However, by performing the global rebuilding incrementally [28, 29] over the next $\Theta(\overline{N})$ updates, this does not increase the asymptotic worst-case number of I/Os of each update. While initializing the new structure there are still updates happening which must then be applied before it can take over. By performing updates to the new structure at a sufficiently fast rate compared to the live structure this ensures that they stay within a constant factor of each other in size until the new structure takes over. Therefore, only the I/O cost of an update without global rebuilding needs to be considered.

Updates are inserted in the root buffer of the B^ε-tree, as described in Section 2.4. By keeping the root buffer in internal memory this uses no I/Os. If the root buffer overflows, it may cause buffer overflows along a root-to-leaf path down to an open rectangle, which may then be finalized by performing an actualize operation followed by a path copy. The update therefore requires $\mathcal{O}(H)$ I/Os in total under the large internal memory assumption. However, each time the root buffer overflows, F updates are removed from it, meaning this occurs at most every Fth update. Thus, when the root buffer overflows, we incrementally apply the update to the structure over the next F updates, ensuring that $\mathcal{O}(H/F)$ I/Os are performed per update in the worst case. To not interfere with the incremental work, we place new updates in a separate buffer while it is in progress and merge them with the root buffer when it is finished. If a path copy has occurred, the root pointers of the F most recent versions must be updated to the new root. Since they are stored together in an array indexed by their version identifier this takes $\mathcal{O}(1)$ I/Os. If a query occurs while an update is

being performed incrementally, we complete the update before executing the query. This does not increase the asymptotic worst-case number of I/Os for queries.

3.2 Smaller Buffers on All Paths Using the Subtraction Game

In the previous section, we described how as long as the internal memory can hold all values on a root-to-leaf path towards the same open rectangle, there is no overhead on worst-case queries and updates compared to the amortized bounds. To lower the possible number of such values, we will slightly change the flushing strategy described in Section 2.4 where we only performed a flush when a buffer overflowed. Instead, for every F'th update, we flush along an entire root-to-leaf path, always flushing towards the child where most of the updates are going. We still flush at most F values, which preserves the property that internal nodes of T contain at most $2\Delta F$ updates. In the following, we show that this flushing strategy guarantees that all buffers contain $\mathcal{O}(F \log_2 \Delta)$ updates going towards the same child, and therefore also the same leaf. This implies that the assumption $M = \Omega(HF \log_2 \Delta) = \Omega\big(B^{1-\varepsilon} \log_2 \overline{N}\big)$ is sufficient to achieve no overhead for worst-case queries and updates. This is a factor $\Theta(B^\varepsilon / \log_2 B)$ improvement over the previous smallest assumption on M [15].

We can view each node as playing the *subtracting game* studied by Dietz and Raman [17] for the number of updates $x_1, x_2, x_3, ..., x_{2\Delta}$ going towards each of its at most 2Δ children. In particular, when at most F updates are flushed towards a node, if there are δ_i new updates going towards the ith child, then variable x_i is increased by δ_i. Then we flush towards the child j where most of the updates are going which sets the variable $x_j = \max\{x_j - F, 0\}$. Following [17, Theorem 3] and Theorem 4 in the Appendix, scaled by a factor of F, this guarantees $x_i = \mathcal{O}(F \log_2 \Delta)$ for any i.

We also need to consider how merging and splitting in T impacts the games. Only leaves of T, corresponding to open rectangles, are merged. When an internal node is split, it corresponds to evenly distributing the x_i variables from one game to two new games, except for one variable that is split into two new variables, each with a smaller or equal value. When a leaf, i.e. an open rectangle, is merged or split, the one or two rectangles involved are first actualized, which sets their variables to zero, a stronger operation than subtracting. Thus, the variable for a new rectangle is always zero and variables on root-to-leaf paths to actualized rectangles may be decremented. In all cases and for all games, variables are either decremented without adding to the game, or a copy of an existing game is created, where all variables in the copy are equal or smaller in value than before. This concludes the proof of Theorem 2.

3.3 Improving Worst-Case RANGE Queries Using the Zeroing Game

In this section, we consider the small-memory setting with $M \geq 2B$, to overcome the theoretical limitation of the memory assumptions made in Sections 3.1 and 3.2. Actualizing a rectangle by merging the relevant updates on a root-to-leaf path to a rectangle requires $\mathcal{O}(H + \gamma)$ total I/Os, where $\gamma = \mathit{sort}\big(B^{1-\varepsilon} \log_2 \overline{N}\big)$. The construction from the previous section directly results in worst-case SEARCH queries and updates in $\mathcal{O}(H + \gamma)$ and $\mathcal{O}\big(\frac{1}{F}(H + \gamma)\big)$ I/Os, respectively. However, since RANGE queries are performed by repeatedly searching for the $\Theta(1 + K/R)$ rectangles intersecting the query, the worst-case number of I/Os is $\Theta((1 + K/R)(H + \gamma)) = \Theta\Big(H + \gamma + \frac{K}{B}\Big(1 + \frac{\varepsilon \log_2 B}{B^\varepsilon} \log_{M/B}\big(B^{-\varepsilon} \log_2 \overline{N}\big)\Big)\Big)$, notably with a multiplicative non-constant overhead on the reporting term. In this section, we describe how to guarantee RANGE queries in worst-case $\Theta\big(H + \gamma + \frac{K}{B}\big)$ I/Os when $M \geq 2B$.

The worst-case I/O cost of a RANGE query can be improved by merging all the buffered updates to the open rectangles intersecting the query in a top-down, layer-by-layer fashion. That is, by essentially actualizing all the open rectangles intersected by the RANGE query simultaneously. We denote the updates already present in the rectangles by the *partial output list*. Rather than applying the buffered updates to the rectangles, we merge them with the partial output list to obtain the final output. A given query RANGE(v, x, y) reports $\Omega(R)$ values from each intermediate rectangles, i.e., all rectangles intersecting the query except for the two that contain the endpoints x and y. Thus, in $\mathcal{O}((1 + K/R)H + (R + K)/B) = \mathcal{O}(H + K/B)$ I/Os we can find all the relevant rectangles and compute the partial output list. To collect the buffered updates in T for the intermediate open rectangles in sorted order, we merge the updates down layer by layer. We only move down the updates to versions earlier or equal to v since only these can affect the query result. Once obtained, these updates are merged with the partial output list using linear I/Os to report the output of the RANGE query.

The buffered updates are stored in T, which contains the open rectangles at version v_c, however, the RANGE query is on the rectangles present at version v. Let T_v denote the B^ε-tree on open rectangles at version v, i.e., the state of T when version v was created. From T_v to T the tree may have changed, but no later updates are relevant for the query. Thus, the total number of relevant updates does not increase from T_v to T, and each update remains on the root-to-leaf path towards the open rectangle to which the update is relevant. The relevant updates in T can be collected in sorted lists ordered by layer by traversing each root-to-leaf path in T towards open rectangles intersecting the query using $\mathcal{O}((1 + K/R)H)$ I/Os. To bound the I/Os to move the updates down layer by layer, we show that the total number of updates is $\mathcal{O}\big(B^{1-\varepsilon} \log_2 \overline{N} + K/H\big)$. To this end, we need the additional invariant that all nodes of degree one have empty buffers, which we show how to obtain below. Let T'_v be the subtree of T_v consisting of all nodes on root-to-leaf paths to rectangles intersecting the query. Then split T'_v into two root-to-leaf paths p_x and p_y to x and y, respectively, along with all the subtrees hanging off p_x or p_y. For a node on p_x (symmetrically p_y) of degree at least two there may be one or more subtrees T_{sub} hanging off the node. Since only the nodes of T_{sub} with degree at least two have non-empty buffers, if T_{sub} has ℓ leaves, the number of buffered updates in T_{sub} is at most $\mathcal{O}((\ell - 1)B)$. Thus, the number of buffered updates in T'_v excluding degree one nodes on p_x and p_y is $\mathcal{O}(K/H)$. A degree one node on p_x and p_y may have a large degree in T_v, but since it only has one child in the direction of the query, due to the subtracting game, it stores at most $\mathcal{O}(F \log_2 \Delta)$ relevant updates. The number of degree one nodes on p_x and p_y is at most $2H$, and they together contribute $\mathcal{O}\big(B^{1-\varepsilon} \log_2 \overline{N}\big)$ buffered updates, which we locate and sort separately using $\mathcal{O}(H + \gamma)$ I/Os. For the remaining $\mathcal{O}(K/H)$ buffered updates, we merge them layer by layer using $\mathcal{O}(H + K/B)$ I/Os. In total, the worst-case number of I/Os to perform a RANGE query is $\mathcal{O}(H + \gamma + K/B)$ I/Os.

Empty Buffers for Degree one Nodes. To ensure that each node of degree one has an empty buffer, we alter the buffer capacity of nodes to scale with the degree. Let the capacity of the buffer of a node with degree $d \geq 2$ be at most $F \cdot \min\{2d, 2\Delta\}$, with nodes of degree one having a buffer capacity of 0. When flushing a node, as the maximum number of updates in the buffer scales with the degree, then at least F updates going to the same child can be found when overflowing. When splitting a node, it must have degree 2Δ, resulting in the two new nodes having degree Δ, which therefore does not decrease the buffer capacity, and flushing is not needed. When a child of a node is removed due to merging rectangles, the degree of the node is decreased by one. If the degree remains at least two, at most two flushes

are required to get the buffer capacity within bounds. Otherwise, if the degree drops from two to one, at most four flushes are needed. To avoid cascading merges of rectangles, we do not finalize a rectangle once it receives a certain number of updates. Instead, we finalize the rectangle that has received the most updates, provided it has received at least R updates. This last condition ensures a bound on the space usage. Including the initial flush from the root buffer, then when a rectangle is finalized, at most $5F$ updates have been flushed into open rectangles.

Let U_i denote the number of updates to the ith rectangle, excluding the initial insertions. We extend the data structure to include an array over all open rectangles, where index j stores a blocked-linked-list of all rectangles where the number of updates is $U_i = j$. Each rectangle has a double linked pointer between its location in the array of lists and the rectangle. This allows for moving a rectangle to a new entry in the array, when it receives updates, as well as finding a rectangle which have received the most updates, by scanning the list.

To show that U_i is bounded by $\mathcal{O}(R)$, we apply the *zeroing game* of Dietz and Sleator [18], using the variables $x_i = \max\left\{0, \frac{U_i - R}{5F}\right\}$ if rectangle i is open and $x_i = 0$ if it is closed. For open rectangles, x_i count the number of units of $5F$ updates received beyond the first R updates. This ensures that the variables are incremented by at most 1 in total for each round, when at most 5 flushes of size at most F are flushed into the open rectangles. When finalizing a rectangle it becomes closed, which ensures that $x_i = 0$, matching the zeroing step. We bound the total number of rectangles by $\overline{N}$ and therefore also the number of variables. Following [18, Theorem 5] and a proof similar to Theorem 4 in the Appendix, we have that for any i then $x_i \leq \log_2 \overline{N} + 1$ at any time. Consequently, it follows that $U_i \leq 5F(\log_2 \overline{N} + 1) + R$. It can be shown that $5F(\log_2 \overline{N} + 1) \leq 2R$, when $\overline{N} \geq 8$, by unfolding R and simplifying the inequality to show that $\frac{\varepsilon \log_2 B}{B^\varepsilon}\left(1 + \frac{1}{\log_2 \overline{N}}\right) \leq \frac{4}{5}$. It therefore holds that each rectangle receives at most $U_i \leq 3R$ updates, due to the zeroing game. Thus, when finalizing a rectangle, at least R updates have been performed. Including the updates from the buffers on the path towards the rectangle, the total number of updates applied is between R and $4R$. By ensuring that each rectangle contains $[8R, 16R[$ initial values, the rebalancing operations are possible, and the spanning criteria remains satisfied.

An update therefore performs at most 5 flushes using $\mathcal{O}(H)$ I/Os, along with locating and finalizing a single rectangle in respectively $\mathcal{O}(R/B) = \mathcal{O}(H)$ and $\mathcal{O}(H + \gamma)$ I/Os. Performing this operation incrementally allows for updates to spend worst-case $\mathcal{O}\left(\frac{1}{F}(H + \gamma)\right)$ I/Os. If a query happens while an incremental update is being performed, the incremental update is completed, using at most $\mathcal{O}(H + \gamma)$ I/Os, which does not increase the total cost of the query. When a SEARCH query happens, similar to the new RANGE query, we do not apply the relevant updates on the path to the open rectangle to avoid queries interfering with the zeroing game. This concludes the proof of Theorem 3.

4 Discussion and Open Problems

Global rebuilding, as described in Section 2.6, allows for constructing a partially-persistent set of any sorted set in a linear number of I/Os, without creating the set anew by a sequence of insertions. Symmetrically, it is possible to *purge* all versions of the set older than some threshold, without performing all updates anew. This problem was first motivated by Becker, Gschwind, Ohler, Seeger, and Widmayer [6]. As our data structure consists of multiple independent data structures covering disjoint version intervals, then all data structures which only cover versions to be purged can be removed efficiently by a linear number of I/Os. For

both use cases, the space usage is asymptotically linear in the size of the oldest stored set and the number of updates performed.

Further, global rebuilding allows for a crude fully persistent data structure, which supports efficient buffered updates and queries, but where cloning past versions requires a linear number of I/Os. The fully persistent data structure by Brodal, Rysgaard, and Svenning [12] allows cloning past versions in worst case $\mathcal{O}(1)$ I/Os. They do, however, not buffer updates, which therefore are amortized and a factor $\mathcal{O}\big(1/B^{1-\varepsilon}\big)$ slower than our data structure. Our data structure is therefore better when there are many updates, but few clones of past versions happening. Further, our data structure is simpler. It remains an open problem to design buffered fully-persistent search-trees, which remains efficient for clone operations.

In Section 3 we showed how to achieve worst-case bounds matching those of ephemeral B^ε-trees, when $M = \Omega\big(B^{1-\varepsilon}\log_2 N\big)$. This is an improvement by a factor $\Theta(B^\varepsilon/\log_2 B)$ on the required lower bound on M over the worst-case results of B^ε-trees by Das, Iacono, and Nekrich [15]. It remains an open problem to show a worst-case I/O lower bound dependency on M or to find a structure with worst-case I/O guarantees matching the amortized I/O bounds for $M = 2B$.

References

1. Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116—1127, September 1988. doi:10.1145/48529.48535.
2. Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1—24, September 2003. doi:10.1007/s00453-003-1021-x.
3. Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8:1.2–es, December 2004. doi:10.1145/996546.996549.
4. Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47:1–25, 2007. doi:10.1007/s00453-006-1208-z.
5. Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.
6. Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996. doi:10.1007/s007780050028.
7. Michael A. Bender, Rathish Das, Martín Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing without cascades. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 650–669. Society for Industrial and Applied Mathematics, 2020. doi:10.1137/1.9781611975994.40.
8. Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 81—-92. Association for Computing Machinery, 2007. doi:10.1145/1248377.1248393.
9. Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to B-trees and write-optimization. *;login:*, 40(5), 2015. URL: https://www.usenix.org/publications/login/oct15/bender.
10. Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages

546—554. Society for Industrial and Applied Mathematics, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644201.

11 Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, pages 307—-315. Association for Computing Machinery, 2003. doi:10.1145/780542.780589.

12 Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. External memory fully persistent search trees. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, pages 1410–1423. Association for Computing Machinery, 2023. doi:10.1145/3564246.3585140.

13 Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 859—860. Society for Industrial and Applied Mathematics, 2000.

14 Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

15 Rathish Das, John Iacono, and Yakov Nekrich. External-memory dictionaries with worst-case update cost. In Sang Won Bae and Heejin Park, editors, *33rd International Symposium on Algorithms and Computation, ISAAC 2022, December 19-21, 2022, Seoul, Korea*, volume 248 of *LIPIcs*, pages 21:1–21:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ISAAC.2022.21.

16 Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Trans. Algorithms*, 3(2):13–es, May 2007. doi:10.1145/1240233.1240236.

17 Paul F. Dietz and Rajeev Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications (abstract). In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 1993. doi:10.1007/3-540-57155-8_256.

18 Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 365–372. Association for Computing Machinery, 1987. doi:10.1145/28395.28434.

19 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

20 Herbert Edelsbrunner and Mark H. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6(4):515–542, 1985.

21 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1), January 2012. doi:10.1145/2071379.2071383.

22 Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57. Association for Computing Machinery, 1984. doi:10.1145/602259.602266.

23 Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/O-efficient generation of massive graphs following the LFR benchmark. *ACM Journal of Experimental Algorithmics*, 23, August 2018. doi:10.1145/3230743.

24 William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin

Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage*, 11(4), November 2015. doi:10.1145/2798729.

25 Curtis P. Kolovson and Michael Stonebraker. Indexing techniques for historical databases. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 127–137. IEEE Computer Society, 1989. doi:10.1109/ICDE.1989.47208.

26 Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Physical Review E*, 80:016118, July 2009. doi:10.1103/PhysRevE.80.016118.

27 Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78:046110, October 2008. doi:10.1103/PhysRevE.78.046110.

28 Mark H. Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1983. doi:10.1007/BFb0014927.

29 Mark H. Overmars and Jan van Leeuwen. Dynamization of decomposable searching problems yielding good worsts-case bounds. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 1981. doi:10.1007/BFB0017314.

30 Rajeev Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time*. PhD thesis, Department of Computer Science, University of Rochester, 1992.

31 Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669—679, July 1986. doi:10.1145/6138.6151.

32 Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997. doi:10.1109/69.599929.

A Subtraction Game

Dietz and Raman [17] considered the following *subtraction game*. The game is played on n non-negative real variables $x_1, \ldots, x_n$, initially all zero. The game progresses in *rounds*. Each round consists of an *increment step* followed by a *subtraction step*. In the increment step an adversary selects n non-negative real values $\delta_1, \ldots, \delta_n$, where $\sum_{i=1}^{n} \delta_i \leq 1$, and sets $x_i \leftarrow x_i + \delta_i$. In the subtraction step a largest x_i is decremented by setting $x_i \leftarrow \max\{0, x_i - 1\}$. Dietz and Raman [17, Theorem 3] proved an upper bound on all variables of $1 + \ln n$. Below we improve this bound to be $H_{n-1} \leq 1 + \ln(n-1)$. Note that $H_{n-1} \leq \log_2 n$ for $n \geq 1$. The proof extends to the *zeroing game* of Dietz and Sleator [18, Theorem 5], where the subtraction step is replaced by a *zeroing step*, in which a largest x_i is set to 0. For both games, it holds that for all i, $x_i < H_{n-1} + 1$ after each step.

▶ **Theorem 4.** *The subtraction game on $n \geq 2$ variables guarantees all $x_i < H_{n-1} \leq 1 + \ln(n-1)$ after each round.*

Proof. For $1 \leq k \leq n$, let s_k denote a strict upper bound on the sum of the k largest variables in the game after any round. The goal is to find a small valid s_1. Below we argue that $s_n = n - 1$ and $s_k = \frac{k}{k+1}(1 + s_{k+1})$, for $1 \leq k < n$, are valid upper bounds. By induction for decreasing k, we have $s_k \geq k$ for $1 \leq k < n$: $s_{n-1} = \frac{n-1}{n}(1 + s_n) = \frac{n-1}{n}(1 + n - 1) = n - 1$ and $s_k = \frac{k}{k+1}(1 + s_{k+1}) \geq \frac{k}{k+1}(1 + k + 1) > k$ for $1 \leq k < n - 1$. By induction for increasing k, we have $s_1 = \sum_{j=2}^{k} \frac{1}{j} + \frac{1}{k} s_k$ for $2 \leq k \leq n$, i.e., $s_1 = \sum_{j=2}^{n} \frac{1}{j} + \frac{n-1}{n} = \sum_{j=1}^{n-1} \frac{1}{j} = H_{n-1}$.

Initially, all x_i are zero, i.e., all $s_1, \ldots, s_n$ are valid strict upper bounds. By induction on the number of rounds, we show that the upper bounds remain valid after each round.

Consider s_n, and assume there exists a round where the total sum is $< n-1$ before the round and $\geq n-1$ after the round. Since the total sum increases, the subtraction step must have decreased the total sum by < 1. It follows that all variables after the increment step must be < 1, and the subtraction step sets one variable to zero, i.e., the total sum after the increment step is $< n-1$. This contradicts the assumption that the total sum is $\geq n-1$ after the round, i.e., the total sum after each round is $< n-1$.

Next, consider s_k, for $1 \leq k < n$. We first consider the case where the subtracted variable $x_i < 1$ before the subtraction step, i.e., all variables are < 1. Then, after the subtraction step the sum of the k largest variables is $< k \leq s_k$. Otherwise, $x_i \geq 1$ after the increment step. If the variable x_i is among the k largest variables after the round, then the sum of the k largest variables can increase by at most $\sum_{j=1}^{n} \delta_j - 1 \leq 0$ in the round (the sum of the k largest variables after the round consists of the same variables as before the round, or variables with smaller value before the round), i.e., the sum of the k largest variables does not increase by the round. Otherwise, x_i is not among the k largest variables after the round, but x_i is the largest after the increment step, where the $k+1$ largest variables have sum $< \sum_{j=1}^{n} \delta_j + s_{k+1}$. After subtracting x_i (i.e., after the round), the sum of the new k largest variables is $< \frac{k}{k+1}\big(\sum_{j=1}^{n} \delta_j + s_{k+1}\big) \leq \frac{k}{k+1}(1 + s_{k+1}) = s_k$. ◀

That the analysis is tight follows from the following strategy, as also described in the PhD thesis of Raman [30, Section 2.2.3]. We first perform sufficiently many initial rounds, where after r rounds one variable has value zero, say $x_1 = 0$, and all other $n-1$ variables have value $\varepsilon_r = 1 - \big(1 - \frac{1}{n}\big)^r$. Note $\varepsilon_0 = 0$ and $\varepsilon_r \to 1^-$ for $r \to \infty$. In round r we let $\delta_1 = \varepsilon_{r-1} + (1 - \varepsilon_{r-1})/n$ and $\delta_2 = \cdots = \delta_n = (1 - \varepsilon_{r-1})/n$. This ensures that the increment step makes all n values have value $\varepsilon_r = \varepsilon_{r-1} + (1 - \varepsilon_{r-1})/n$ before the subtraction step. By induction it follows that $\varepsilon_r = 1 - \big(1 - \frac{1}{n}\big)^r$. By performing a sufficient number of initial rounds, $n-1$ variables can achieve value $1 - \varepsilon$ arbitrary close to one. In the next $n-2$ rounds, for $j = n-1, \ldots, 2$, we distribute value $1/j$ to the j variables with maximum value $1 - \varepsilon + \sum_{i=j+1}^{n-1} \frac{1}{i}$. The final maximum value is $1 - \varepsilon + \sum_{i=2}^{n-1} \frac{1}{i} = H_{n-1} - \varepsilon$.

Chapter 7

External Memory Fully Persistent Search Trees

External Memory Fully Persistent Search Trees*

Gerth Stølting Brodal
Department of Computer Science
Aarhus University
Denmark
0000-0001-9054-915X
gerth@cs.au.dk

Casper Moldrup Rysgaard
Department of Computer Science
Aarhus University
Denmark
0000-0002-3989-123X
rysgaard@cs.au.dk

Rolf Svenning
Department of Computer Science
Aarhus University
Denmark
0000-0002-9903-4651
rolfsvenning@cs.au.dk

July 31, 2025

Abstract

We present the first fully-persistent external-memory search tree achieving amortized I/O bounds matching those of the classic (ephemeral) B-tree by Bayer and McCreight. The insertion and deletion of a value in any version requires amortized $\mathcal{O}(\log_B N_v)$ I/Os and a range reporting query in any version requires worst-case $\mathcal{O}(\log_B N_v + K/B)$ I/Os, where K is the number of values reported, N_v is the number of values in the version v of the tree queried or updated, and B is the external-memory block size. The data structure requires space linear in the total number of updates. Compared to the previous best bounds for fully persistent B-trees [Brodal, Sioutas, Tsakalidis, and Tsichlas, SODA 2012], this paper eliminates from the update bound an additive term of $\mathcal{O}(\log_2 B)$ I/Os. This result matches the previous best bounds for the restricted case of partial persistent B-trees [Arge, Danner and Teh, JEA 2003]. Central to our approach is to consider the problem as a dynamic set of two-dimensional rectangles that can be merged and split.

Keywords: B-trees, range queries, external memory, full persistence, semi-dynamic ray shooting.
CCS concepts: Theory of computation – Data structures design and analysis.

1 Introduction

The B-tree of Bayer and McCreight [6] is the classic external-memory data structure for storing a dynamic set $\mathcal{S}$ of N totally ordered values. It supports the insertion and deletion of values in $\mathcal{O}(\log_B N)$ I/Os and range reporting queries in $\mathcal{O}(\log_B N + K/B)$ I/Os, i.e., reporting all values contained in a value range $[x, y]$, where K is the number of values reported and B is the external-memory block size.

In this paper, we consider the problem of storing a dynamic set in external memory *fully persistently*, i.e., all versions of the set are remembered, any previous version can be queried, and any previous version can be updated resulting in a new version of the set. We present the first fully persistent search trees matching the asymptotic I/O bounds of the classic *ephemeral* (i.e., non-persistent) B-trees in the amortized sense.

Throughout this paper, we will assume the I/O model of Aggarwal and Vitter [2] consisting of an internal memory being able to hold M values, and an infinite external memory. One I/O can transfer B consecutive values between internal and external memory, and the I/O complexity of an algorithm is the number of I/Os performed. Computation can only be performed in internal memory, and the only allowed operations on values are comparisons.

*Work supported by Independent Research Fund Denmark, grant 9131-00113B. A short version of this paper appears in the proceedings of the 55th ACM Symposium on Theory of Computing (STOC 2023) [11].

Table 1: I/O bounds for previous and our results for ephemeral, partially and fully persistent search trees in external memory. All structures use linear space.

	Reporting	Update
Ephemeral		
Bayer and McCreigh[6]	$\mathcal{O}(\log_B N + K/B)$	$\mathcal{O}(\log_B N)$
Brodal and Fagerberg[10]	$\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N + K/B\big)$	$\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N\big)$ am.
Partial persistence		
Lomet and Salzberg [24]	$\mathcal{O}(N_v/B)$	$\mathcal{O}(\log_B N_v \log_B N)$
Becker, Gschwind, Ohler, Seeger, Widmayer [7]	$\mathcal{O}(\log_B N + K/B)$	$\mathcal{O}(\log_B N_v)$ am.
Varman and Verma [27]		
Arge, Danner, Teh [4]	$\mathcal{O}(\log_B N + K/B)$	$\mathcal{O}(\log_B N)$
Full persistence		
Lanka and Mays [23]	$\mathcal{O}((\log_B N_v + K/B)\log_B N)$	$\mathcal{O}\big(\log_B^2 N_v\big)$ am.
Brodal, Sioutas, Tsakalidis, Tsichlas [12]	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v + \log_2 B)$ am.
This paper	$\mathcal{O}(\log_B N_v + K/B)$	$\mathcal{O}(\log_B N_v)$ am.

1.1 Interface of a Fully Persistent Search Tree

The interface to a fully persistent search tree consists of the below operations. Each version is identified by a unique version identifier v (a positive integer). We let $\mathcal{S}_v$ denote the set of values at version v. The versions form a *version tree* T, where the root (version 1) stores the initial set, and a new version w becomes a child of an existing version v in T, if $\mathcal{S}_w$ is derived as a *clone* of $\mathcal{S}_v$. Updates (insertions and deletions) can only be applied at leaves of the version tree T, i.e., versions that have not been cloned yet. These versions are said to be *unlocked*. A version that has been cloned cannot be updated further and is said to be *locked*. All versions can be queried and cloned. If one needs to update a locked version (an internal node of the version tree), one has to clone the version and apply the update to the new unlocked version (leaf of the version tree).

CLONE(v) Creates a new version w, where $\mathcal{S}_w = \mathcal{S}_v$. Returns the new version identifier w. Version w becomes a child of v in the version tree T, i.e., after the operation version v is locked and version w is unlocked.

INSERT(v, x) Adds x to $\mathcal{S}_v$. Requires version v is unlocked.

DELETE(v, x) Removes x from $\mathcal{S}_v$. Requires version v is unlocked. If x is not contained in $\mathcal{S}_v$, nothing is changed.

RANGE(v, x, y) Returns all values in $\mathcal{S}_v \cap [x, y]$ in increasing order.

SEARCH(v, x) Returns the predecessor of x in $\mathcal{S}_v$.

We let N_v denote the number of values in $\mathcal{S}_v$, and N the total number of updates done to all versions. Figure 1 (left) illustrates a version tree. In the following examples, we illustrate values as integers, but our construction works for any comparison based values.

1.2 Previous Work

For the non-persistent setting in internal memory, a set can be stored using a standard balanced binary search tree, e.g., AVL-trees [1], red-black trees [21], and (a, b)-trees [22]. These all support insertions and deletions in $\mathcal{O}(\log_2 N)$ time, and range reporting queries in $\mathcal{O}(\log_2 N + K)$ time. In external memory, the B-tree [6] is the classic data structure of choice and supports updates in $\mathcal{O}(\log_B N)$ I/Os and range reporting queries in $\mathcal{O}(\log_B N + K/B)$ I/Os. A number of papers have since explored the update-query trade-off starting with [10] giving a structure known as the B^ε-tree for any $0 < \varepsilon < 1$ with amortized $\mathcal{O}\big(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N\big)$ I/Os for updates and $\mathcal{O}\big(\frac{1}{\varepsilon}\log_B N\big)$ I/Os for queries. The B^ε-tree operates similarly to a B-tree but with fanout B^ε and updates are flushed down the tree using buffers, leading to the amortized bound. In [9, 8] the amortized bound was improved to high probability and finally in [16] to worst-case.

The notion of (data structural) full persistence for internal memory data structures was coined by Driscoll, Sarnak, Sleator and Tarjan in [19], where they also described how to achieve a fully persistent

version of red-black trees. Previous to this Sarnak and Tarjan [26] had presented partially persistent red-black trees, where partial refers to the limitation that only the most recent version of the red-black tree can be updated, i.e., all versions of the red-black tree form a version tree consisting of a single path with the most recent version of the red-black tree being the single leaf. Demaine, Iacono and Langerman [17] considered the notion of *retroactive* data structures, where updates can be applied to any version in the version tree, and updates are automatically recursively applied to all derived version.

Adopting the notions of partial and full persistence to external-memory search trees has been done in a sequence of papers. See Table 1. The results of Arge, Danner and Teh 2003 [4] and Brodal, Sioutas, Tsakalidis and Tsichlas [12, 13] achieve the best bounds for partial and full persistence, respectively. Both results achieve optimal bounds for range reporting, i.e., $\mathcal{O}(\log_B N_v + K/B)$ I/Os (the N in the bounds of [4] for partial persistence can be reduced to N_v by applying global rebuilding after a linear number of updates). The update I/O bound of [4] matches the $\mathcal{O}(\log_B N)$ I/Os for B-trees, whereas the full persistence result of [12] has an additive $\mathcal{O}(\log_2 B)$ I/Os overhead per update. This paper eliminates this additive term and achieves update bounds matching those of (ephemeral) B-trees and [4] for partial persistence.

1.3 Contribution

The main contribution of this paper are fully-persistent external-memory search trees, achieving the following bounds, matching those of (ephemeral) external-memory B-trees.

Theorem 1. *There exist external-memory fully-persistent search trees supporting* INSERT *and* DELETE *in amortized* $\mathcal{O}(\log_B N_v)$ *I/Os,* CLONE *in worst-case* $\mathcal{O}(1)$ *I/Os,* SEARCH *in worst-case* $\mathcal{O}(\log_B N_v)$ *I/Os, and* RANGE *in worst-case* $\mathcal{O}(\log_B N_v + K/B)$ *I/Os. The space usage is linear in the number of updates.*

It should be noted that for the case where $N_v = B^{\mathcal{O}(1)}$, the update bounds are $\mathcal{O}(\log_B N_v) = \mathcal{O}(1)$ I/Os, whereas the best previous update bound was $\mathcal{O}(\log_2 B) = \mathcal{O}(\log_2 N_v)$ I/Os [12]. Our range reporting queries can be extended to support an argument k, and to only report in sorted order the first k (or last k) values in a value range $[x, y]$ using $\mathcal{O}(\log_B N_v + k/B)$ I/Os, by simply truncating the reporting when k values have been found.

The basic ideas of the construction are the following. As in [19], we use a linearization of the version tree into a version list obtained by a pre-order traversal of the version tree. Values are associated with half-open liveness intervals of the version list. See Figure 1 (right). To adapt this to external memory setting, we consider these (value, liveness interval) pairs as vertical segments in a two-dimensional plane, and partition the plane into rectangles, possible splitting segments into multiple smaller disjoint segments. See Figure 2. During insertions and deletions the partition into rectangles is dynamically updated by splitting rectangles in either the version dimension (corresponding to the node splitting idea in [19]) or value dimension (corresponding to splitting a leaf in a B-tree), or by joining two horizontally adjacent rectangles (corresponding to joining two adjacent leaves in a B-tree). See Figure 3. A range reporting query $\textsc{Range}(v, x, y)$ simply identifies all rectangles intersected by the horizontal query segment $[x, y] \times \{v\}$, and reports the values of all vertical segments in these rectangles intersecting the horizontal query segment. Invariants ensure that for all rectangles intersected, except for two, a constant fraction of the values are part of the answer to the query. This is inspired by the filtering search of Chazelle [14].

Essential to our result is the application of a two-dimensional orthogonal point location (vertical ray shooting) structure to identify the rectangle containing a (value, version) point. Point location queries must be supported in worst-case $\mathcal{O}(\log_B N)$ I/Os, whereas segment insertions are allowed to be significantly slower, as high as amortized $\mathcal{O}(B \log_B^2 N)$ I/Os. The best external-memory bounds for dynamic orthogonal point location are by Munro and Nekrich [25], who support updates and queries with $\mathcal{O}(\log_B N \log\log_B N)$ I/Os, and by Arge, Brodal and Rao [3], who support queries with $\mathcal{O}(\log_B^2 N)$ I/Os and updates with $\mathcal{O}(\log_B N)$ I/Os (in internal memory $\mathcal{O}(\log n)$ time updates and queries were achieved by Giora and Kaplan [20]). To achieve queries with $\mathcal{O}(\log_B N)$ I/Os, we describe a specialized insertion-only external-memory point location structure, that also crucially avoids the comparison of version identifiers. The internal-memory [19] and external-memory [12] fully persistent search trees make essential use of the dynamic order-maintenance data structure by Dietz and Sleator [18] to answer order queries among two version identifiers in the version list in $\mathcal{O}(1)$ time and I/Os. This is in fact the bottleneck causing the additive $\mathcal{O}(\log_2 B)$ I/Os on updates in [12]. We avoid the use of [18] by using an external-memory colored predecessor data structure, that essentially stores multiple predecessor data structures in a single B-tree.

Table 2: I/O bounds of the operations for each improvement of the structure. All structures use linear space.

	Clone	Insert & Delete	Search	Range
Geometric Structure (Sections 2, 3, 4, 5)	$\mathcal{O}(\log_B \overline{N})$	$\mathcal{O}(\log_B \overline{N})$	$\mathcal{O}(\log_B \overline{N})$	$\mathcal{O}(\log_B \overline{N} + K/B)$
Partitioning the Version Tree (Section 6)	$\mathcal{O}(\log_B N_v)$	$\mathcal{O}(\log_B N_v)$	$\mathcal{O}(\log_B N_v)$	$\mathcal{O}(\log_B N_v + K/B)$
Lazy Clones (Section 7)	$\mathcal{O}(1)$	$\mathcal{O}(\log_B N_v)$	$\mathcal{O}(\log_B N_v)$	$\mathcal{O}(\log_B N_v + K/B)$

1.4 Outline of Paper

In Section 2 we present our geometric interpretation of the fully persistence search tree problem and describe our data structure for the static case. In Sections 3–5 we present a simplified dynamic result, Theorem 2 below, where we assume an upper bound $\overline{N}$ of the total number of updates to be performed and use $\overline{N}$ as a parameter in our construction, and Clone uses $\mathcal{O}(\log_B \overline{N})$ I/Os. In Section 3 we describe the dynamic version of our data structure and in Sections 4 and 5 we describe our colored predecessor and point location structures, respectively. In Sections 6–7 we then show how to improve these bounds to those of Theorem 1. In Section 6 we show how to make the I/O bounds depend on the size N_v of a version v, instead of the upper bound parameter $\overline{N}$, by splitting the version tree into multiple version trees, where all versions in a version tree have approximately equal sizes. In Section 7 we show how to reduce the cost of Clone from $\mathcal{O}(\log_B N_v)$ I/Os to $\mathcal{O}(1)$ I/Os by performing *lazy clones*, i.e., the actual cloning is postponed until the first update is performed on a version. An overview of the I/O bounds for each of these improvements is presented in Table 2.

Theorem 2. *Let $\overline{N}$ be a parameter giving an upper bound on the total number of updates. Then there exist external-memory fully-persistent search trees that support* Insert, Delete *and* Clone *in amortized $\mathcal{O}(\log_B \overline{N})$ I/Os,* Search *in worst-case $\mathcal{O}(\log_B \overline{N})$ I/Os, and* Range *in worst-case $\mathcal{O}(\log_B \overline{N} + K/B)$ I/Os. The space usage is linear in the number of updates.*

2 Static Data Structure

Our implementation of a persistent search tree takes a geometric approach. In Section 2.1 we introduce the version tree and version list, concepts that we borrow from [19], and in Section 2.2 we give a geometric interpretation of range reporting queries. In Section 2.3 we introduce our rectangular two-dimensional space partition in the static setting, and in Section 2.4 we summarize the main properties of the rectangular partition that should be maintained by the dynamic structure in Section 3. In Section 2.5 we describe how to support queries at the high level.

2.1 Version Tree and Version List

Following [19], we define the version tree T as follows. The root is version 1. Whenever a new version w is created, it is assigned the smallest unused positive integer value so far. If w is created by cloning v, then w becomes the leftmost child of v. This approach guarantees if T contains s versions, then these are versions $\{1, 2, \ldots, s\}$, that version identifiers are unique, the version identifiers of the children of a node are increasing from right to left, and the version identifiers along a root-to-leaf path are increasing.

Similarly to [19], we derive a *version list* L from T by a preorder left-to-right traversal of T. As version 1 is the root of the version tree T, it is also the first version in L. We assume that L is terminated by version 0. Given two versions v and w, we in the following let $[v, w[$ denote the versions in the version list from v up to w, but excluding w. Note that in the version list $[1, 0[$ includes all nodes of the version list, except for the terminating version 0. With each value x inserted into version v we store the *liveness interval* $[v, \text{succ}(v)[$, where $\text{succ}(v)$ is the successor of v in the version list. If version w is created by cloning version v, then w becomes the leftmost child of v in the version tree, i.e., w should be inserted immediately after v in the version list. Crucial to the definition of the liveness interval, which is also used in [19], is that the later versions w created below v in the version tree are exactly the versions that will be contained in the liveness interval of x, if x was inserted in version v.

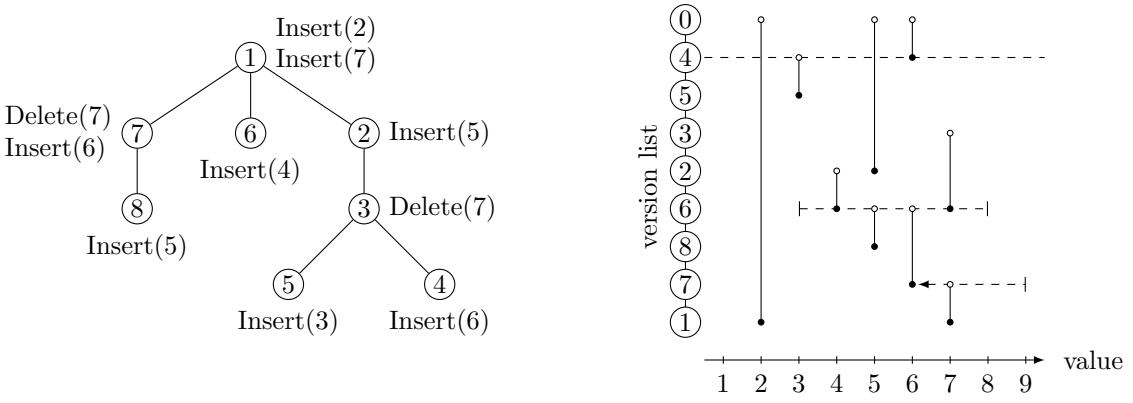


Figure 1: (Left) Version tree illustrating 8 versions of a set after $N = 10$ updates. Versions 1, 2, 3, and 4 contain the sets $\{2, 7\}$, $\{2, 5, 7\}$, $\{2, 5\}$, $\{2, 5, 6\}$, respectively. Updates to a version are shown next to the node of the version tree. (Right) The version list, consisting of a left-to-right preorder traversal of the version tree terminated by 0, and the half-open liveness intervals of versions containing a value. Note that the value 7 only is contained in versions 1, 6 and 2 of the set. The topmost dashed line shows that version 4 of the set contains $\{2, 5, 6\}$, the dashed line segment at version 6 shows that the range reporting query RANGE$(6, 3, 8)$ has result $\{4, 7\}$, and the bottommost dashed arrow that the query SEARCH$(7, 9)$ has result 6.

In Figure 1 (right) the liveness intervals are shown for all values inserted in the example in Figure 1 (left). E.g., the value 2 is inserted into version 1 with liveness interval $[1, 0[$, since only version 1 exists when 2 is inserted. The value 5 is inserted into versions 2 and 8, with liveness intervals $[2, 0[$ and $[8, 6[$, respectively (versions 3, 4 and 5 did not yet exist when 5 was inserted into version 2). Finally, value 7 is inserted in version 1 with liveness interval $[1, 0[$ but again deleted in versions 3 and 7 for liveness intervals $[3, 0[$ and $[7, 6[$, respectively. The result is that value 7 only exists for the remaining version intervals $[1, 7[$ and $[6, 3[$, i.e., versions 1, 2 and 6 in the current version tree.

2.2 Geometry of Updates and Queries

Our problem has a natural geometric interpretation in a two-dimensional space, see Figure 1 (right). Consider the plane where x-values correspond to values, and the y-value corresponds to the order in the version list. INSERT(v, x) corresponds to inserting the line segment $\{x\} \times [v, \text{succ}(v)[$, whereas DELETE$(v, x)$ to deleting the line segment $\{x\} \times [v, \text{succ}(v)[$, possibly splitting a longer line segment into two disjoint segments. Crucial for the implementation of deletions is that the length of the interval to be deleted (at the time of deletion) is only between two adjacent versions in the version list. The values in $\mathcal{S}_v$ are exactly those vertical line segments intersecting the horizontal line at version v, and the answer to the range reporting query RANGE(v, x, y) are exactly the vertical segments intersected by $[x, y] \times \{v\}$, and the answer to SEARCH(v, x) is the rightmost vertical segment intersected by $[-\infty, x] \times \{v\}$.

2.3 Static Rectangular Partitioning

Assume N updates have been performed on a fully persistent search tree. In the following, we present a static structure storing the resulting versions using the geometric representation discussed in Section 2.2 and supports queries in $\mathcal{O}(\log_B N)$ I/Os. In Section 3 we extend this structure to support updates.

A key cornerstone of our structure is the parameter

$$R = \Theta(B \log_B N) \ .$$

A crucial property that we will make repeated use of is that, given a list of R segments in external memory stored in $\mathcal{O}(R/B)$ blocks, the list can be scanned using $\mathcal{O}(R/B) = \mathcal{O}(\log_B N)$ I/Os.

We partition the value $\times$ version plane into disjoint rectangles $r_1, r_2, \ldots, r_k$, see Figure 2. We denote the full plane by $[-\infty, +\infty[\times[1, 0[$ (by slight misuse of notation we let $[-\infty, +\infty[$ denote the whole value range), and each rectangle r is of the form $[x^r, y^r[\times[v^r, w^r[$, where $x^r < y^r$ are values in the value range, possibly $x^r = -\infty$ and $y^r = \infty$, and v^r and w^r are versions, where w^r is strictly after v^r in the version list. The vertical segments corresponding to liveness intervals of a value (Section 2.1) are split into shorter

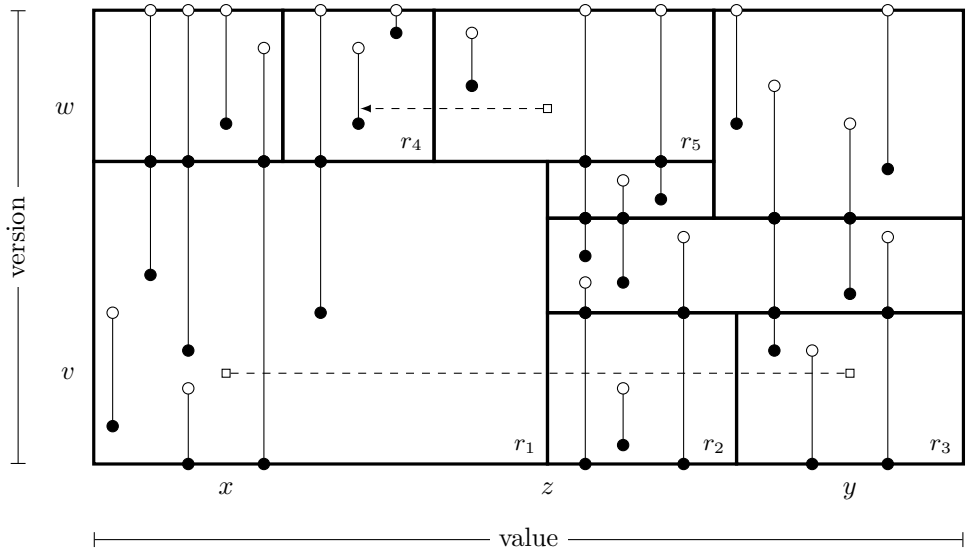


Figure 2: The partitioning of the plane into rectangles, a query RANGE(v, x, y) represented by the dashed line between two square endpoints spanning rectangles r_1, r_2, and r_3, and a query SEARCH(w, z) represented by the dashed arrow originating from a square endpoint, spanning rectangles r_5 and r_4. Note that black dots on vertical segments correspond to the upper endpoint of the segment in the rectangle below and the lower endpoint of the segment in the rectangle above.

segments such that segments never overlap with two rectangles, i.e., a segment $\{x\} \times [v, w[$ is repeatedly split into segments $\{x\} \times [v, v^r[$ and $\{x\} \times [v^r, w[$, if a rectangle $r = [x^r, y^r[\times[v^r, w^r[$ is intersected by the segment, i.e., $x^r \leq x < y^r$ and $v^r \in]v, w[$. For a rectangle r, we let S^r denote the set of vertical segments $\{x\} \times [v, w[$ contained in r.

If the rectangle partition is constructed such that each rectangle contains $\mathcal{O}(R)$ segments, each rectangle is stored in $\mathcal{O}(R/B)$ blocks in external memory, and we have a point location structure that can report the rectangle containing a given a query point using $\mathcal{O}(\log_B N)$ I/Os (e.g., the structure in Section 5), then all segments in the rectangle containing a query point can be reported using $\mathcal{O}(\log_B N + R/B) = \mathcal{O}(\log_B N)$ I/Os.

As seen in Figure 2 a range reporting query may need to report values from multiple rectangles. Assuming that we for each rectangle can output $\Theta(R)$ values, except for the at most two rectangles containing the endpoints of the range reporting query, then each rectangle can be found using $\mathcal{O}(\log_B N)$ I/Os by the point location structure, and the I/O cost for querying each of these rectangles can be charged to the output. The total number of I/Os for a range reporting query becomes $\mathcal{O}(\log_B N + K/B)$ I/Os, where K is the total number of values reported by the range reporting query.

We require that each non-rightmost rectangle contains $\Theta(R)$ segments spanning all versions of the rectangle. We denote these *spanning segments*, i.e., segments $\{x\} \times [v^r, w^r[$ contained in a rectangle $r = [x^r, y^r[\times[v^r, w^r[$, where $x^r \leq x < y^r$. This ensures that all rectangles considered by a SEARCH and RANGE query for a given version contain $\Theta(R)$ values, except for possibly the rightmost rectangle. Note that for SEARCH queries, the predecessor must be reported, and by the spanning requirement, if the predecessor does not exist in the current rectangle, then it must either exist in the left neighboring rectangle for this version, or no predecessor exists. For each rectangle the segments are stored in increasing value order, so that range reporting queries can report values in increasing order.

To be able to efficiently compare versions within a rectangle $r = [x^r, y^r[\times[v^r, w^r[$, we maintain the *local version list* $L^r \subseteq L \cap [v^r, w^r]$ for r, that contains the two versions v^r and w^r and all versions that define endpoints of segments in S^r. Since $|S^r| = \mathcal{O}(R)$, we also have $|L^r| = \mathcal{O}(R)$. The versions in L^r are stored in the same order as in the global version list L. With each version $v \in L^r$ we store the *local version* $\pi^r(v)$ of v in L^r, that just is the position of v in L^r, i.e., an integer in the range $1..|L^r|$. The important property is that v is before w in L^r if and only if $\pi^r(v) < \pi^r(w)$, where the later is a simple comparison between two integers.

We store the segments in S^r only using the local versions. Segment $\{x\} \times [v, w[$ in S^r is stored as the triple $(x, \pi^r(v), \pi^r(w))$, and S^r is stored in increasing value order in $\mathcal{O}(|S^r|/B)$ blocks in external memory. When a new version v is added to L^r, all versions w after v in L^r get their local version $\pi^r(w)$ increased

by one, and similarly all segments in S^r need to be checked if the local version of their endpoints must be increased by one.

If the version v of a query point (x, v) is not the endpoint of any of the segments in the rectangle r containing (x, v), i.e., $v \notin L^r$, let u be the predecessor of v and w the successor of v in the local version list. Since no version between u and w are endpoints of any segments and all segments are inclusive in the first version and exclusive in the second version, all versions between u and w must have the same values in the value range of r. Since $u \in L^r$, we can use $\pi^r(u)$ to compare against the local versions of the segment endpoints in S^r during a query. In Section 4 we describe a structure that can find the predecessor version in a local version list using $\mathcal{O}(\log_B N)$ I/Os, which thus does not increase the I/O cost of queries.

2.4 Structural Requirements

The below list summarizes the main properties required by our rectangular partition from the previous sections. In Section 3 we explain how to maintain them dynamically.

- A rectangle stores $\mathcal{O}(R)$ segments.
- The number of spanning segments in a rectangle is $\Theta(R)$, except for rightmost rectangles.
- The segments are stored in increasing value order.
- For each rectangle and segment in a rectangle we store the local versions.

Additionally, we need two more structures. The first structure we call a *colored predecessor* structure which relates global and local versions. The second structure is a *point location* structure which allows us to navigate the rectangular partitioning. Crucial to both structures is that they should avoid comparing the relative order of versions in the global version list, as we also do internally in the rectangles by using local version lists. That is, we need data structures for the following two problems:

- Colored predecessor problem (Section 4): Given a version v and a rectangle r find the predecessor version u of v in the local version list L^r using $\mathcal{O}(\log_B N)$ I/Os. For this problem, we think of each rectangle as a unique color.
- Point location (Section 5): Given a query point (x, v) find the rectangle containing the query point using $\mathcal{O}(\log_B N)$ I/Os.

2.5 Queries

To perform $\textsc{Range}(v, x, y)$ we first perform a point location query to find the rectangle r containing the point (x, v) and perform a colored predecessor query to find the predecessor u of v in L^r. We find the vertical segments in r intersected by the horizontal segment $[x, y] \times \{u\}$ by scanning through all triples (z, i, j) in S^r and report those z where $x \leq z \leq y$ and $i \leq \pi^r(u) < j$, i.e., the local version interval contains $\pi^r(u)$. If $y^r < y$ we recursively call $\textsc{Range}(v, y^r, y)$. The properties of the partitioning ensure that for each rectangle visited, we will report $\Theta(R)$ values, except possibly the leftmost and rightmost rectangles considered.

To perform $\textsc{Search}(v, x)$, similarly to $\textsc{Range}(v, x, y)$ we first locate the rectangle containing the point (x, v), but now we instead traverse left until the first segment intersection with $[-\infty, x] \times \{v\}$ is identified, which will be the predecessor of x in $\mathcal{S}_v$ (how to move to the rectangle to the left of another rectangle is described in Section 3.2). It is guaranteed that at most two rectangles are required to be scanned due to spanning segments requirement.

Assuming the I/O bounds for the colored predecessor structure in Section 4 and the point location structure in Section 5 are $\mathcal{O}(\log_B N)$, we from the discussion in this section get that $\textsc{Range}$ and $\textsc{Search}$ queries can be performed in $\mathcal{O}(\log_B N + K/B)$ and $\mathcal{O}(\log_B N)$ I/Os, respectively.

3 Updates

In Section 2.3 we described how to efficiently perform queries given a rectangular partition of the plane, assuming some structures exist to report rectangles and version predecessors, where each rectangle had to meet a list of requirements as summarized in Section 2.4. In this section, we discuss how to turn

these requirements into invariants, such that the structure can be made dynamic, i.e., allow for cloning versions and performing insertions and deletions of values in the different versions. We assume, that on initialization we are given a constant $\overline{N}$, which is an upper bound on the total number of updates to be performed on the structure, and let

$$R = B \log_B \overline{N} .$$

When performing a CLONE operation, the geometric view remains the same, apart from a horizontal part being "stretched" to make room for the new version. All information stored in the rectangles remains the same, as the new version is not contained in any segment endpoints. The new version must therefore only be inserted in the global version list L and the colored predecessor structure from Section 4. Assuming that a direct pointer to the version cloned in L is provided, the new version can be inserted into L using worst-case $\mathcal{O}(1)$ I/Os. A new version can be inserted in the colored predecessor structure using amortized $\mathcal{O}(\log_B \overline{N})$ I/Os. In total the CLONE operation requires amortized $\mathcal{O}(\log_B \overline{N})$ I/Os.

The INSERT operation is performed as discussed in Section 2.2 by inserting a value x into the structure at version v is equivalent to adding the segment $\{x\} \times [v, \text{succ}(v)[$. Similarly, to perform a DELETE operation of a value x at version v, then the segment $\{x\} \times [v, \text{succ}(v)[$ is to be removed. Note that this potentially means partitioning an already present segment $\{x\} \times [u, w[$, where $u \leq v < w$, into two new segments, where the piece of the segment between versions v and $\text{succ}(v)$ is removed.

As $\text{succ}(v)$ is the version immediately after v in the version list, and all rectangles are exclusive the top version, then the segment $\{x\} \times [v, \text{succ}(v)[$ exists entirely within one rectangle. This rectangle r can be found as the rectangle containing the point (x, v) using the point location structure of Section 5 with $\mathcal{O}(\log_B \overline{N})$ I/Os. L^r can then be updated with versions v and $\text{succ}(v)$, which are either inserted or deleted to make L^r reflect the current segment endpoints. The locations in L^r can be found using the colored predecessor structure in Section 4 with $\mathcal{O}(\log_B \overline{N})$ I/Os. As new versions are inserted into L^r, this may require the colored predecessor structure to be updated using $\mathcal{O}(\log_B \overline{N})$ I/Os for each newly inserted version. Next, the segments of S^r must be updated for the segment $\{x\} \times [v, \text{succ}(v)[$, as well as the local version of all segments in S^r. Note that the local version of all versions present after v in L^r is incremented or decremented by either 0, 1 or 2, depending on how many versions were inserted or deleted from L^r. The segments of S^r can thus be updated in a single scan using $\mathcal{O}(\log_B \overline{N})$ I/Os by the requirement on the size of S^r.

Updates can thus be performed using $\mathcal{O}(\log_B \overline{N})$ I/Os. They may however break the requirements that were placed on the rectangles to ensure that queries are efficient. In order to make sure that these requirements are met, they are listed in Section 3.1 as invariants, where the asymptotic bounds are replaced by defined constants.

3.1 Invariants

For a rectangle $r = [x^r, y^r[\times [v^r, w^r[$ we define a *spanning segment* to be a segment $\{x\} \times [v^r, w^r[\in S^r$, i.e., the segment represents a value x, which is presented in all versions r spans. The following invariants are then placed on the rectangle:

- I_1: $|S^r| \leq 2R$, i.e., a rectangle stores at most $2R$ segments.
- I_2: The number of spanning segments in r is at least $\frac{R}{4}$, except if $y^r = +\infty$, i.e., r is a rightmost rectangle, where there is no lower bound on the number of spanning segments.
- I_3: The segments in S^r are stored in sorted order by the values the segments represent.
- I_4: The list L^r contains precisely the two versions v^r and w^r, and the versions of all endpoints of segments in S^r.

Invariants I_3 and I_4 can easily be maintained upon an update (see Section 3.7). Invariants I_1 and I_2 may however be broken by an update. For insertions, only invariant I_1 may break, and for deletions, both invariants may break. Note that deletions can break both invariants at once.

In order to handle invariant I_1, the issue is the rectangle now contains too many segments. This can be fixed by repeatably splitting the rectangle into two smaller rectangles, until they all do not contain too many segments. These splits may be done either vertically or horizontally, as illustrated in Figure 3.

The invariant I_2 ensures that each rectangle contains sufficiently many spanning segments. If there are too few spanning segments, denoted as an *underflowing* rectangle, and the rectangle is not a rightmost rectangle, then there must exist some rectangles, which are the right neighbors of the underflowing

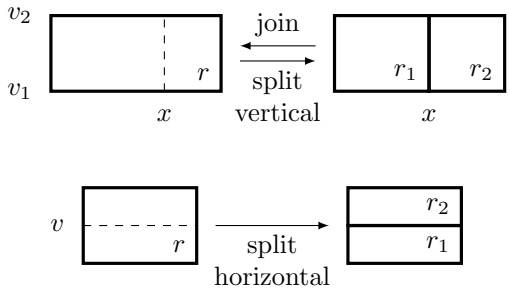


Figure 3: Primitive rectangle transformations: Vertical split and join at value x, and horizontal split at version v.

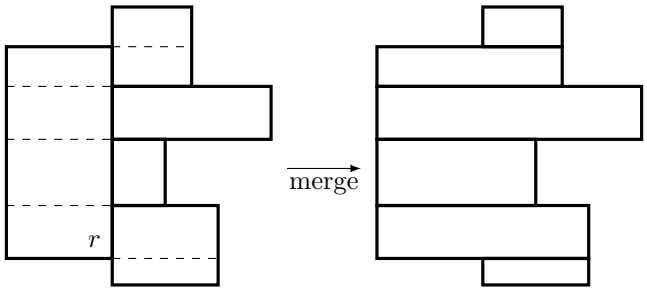


Figure 4: Merging of a rectangle r with its right neighboring rectangles. The dashed lines represent the location that the rectangles are horizontally split, before they are vertically joined.

rectangle. These rectangles must then either contain sufficiently many spanning segments or be rightmost rectangles themselves. By performing horizontal splits to align the rectangles, then the underflowing rectangle can be joined with the right neighbors to become a rightmost rectangles or to contain sufficiently many spanning segments. See Figure 4 for an illustration of how this works. This figure only covers the basic idea of merging, and it later turns out, that more cases for merging are needed, which can be seen in Figure 5. Note that merge is performed by making multiple primitive rectangle transformations.

A merge operation takes time proportional to its number of neighbors. Therefore, we place an upper bound t_N on the number of right neighbors, for some integer constant $t_N \geq 2$. When making a merge operation, the top and bottom right neighbors are split, which if they are close to underflowing will not be an issue, as only one of the two rectangles they are split into will remain close to underflowing. If however there only is one right neighbor, and it is close to underflowing, it may be split twice, and make two rectangles, which are close to underflow. In this case, one of the resulting rectangles may be merged with all of its left neighbors, which is symmetrical to merging to the right, see Figure 5b. Note that there must then also be an upper bound t_N on the number of left neighbors.

This thus adds an extra invariant for each rectangle r, with N_r^R and N_r^L denoting the number of neighbors to the right and left of rectangle r respectively.

I_5: $N_r^R \leq t_N$ and $N_r^L \leq t_N$, i.e., the number of neighboring rectangles to rectangle r is at most t_N on both sides.

Note that horizontal splits may increase the number of neighbors in some rectangles. However, if some rectangle has too many neighbors, then it can be split horizontally, to distribute the number of neighbors between two rectangles, and thus decrease the number of neighbors each rectangle has.

3.2 Finding the Neighbors

For the construction, we must be able to find the neighbors of a given rectangle for merging and detecting neighbor overflow. To find the neighbors to the right of a rectangle, we use that the rectangles are exclusive in the top and right coordinates. By repeatedly querying the endpoints using the point location structure from Section 5, all right rectangles can be found from the bottom up. To find the left rectangles, we need a point inside the left neighbor. Such a point can be found by maintaining a search tree over all rectangle start points and searching for the predecessor of the current left side of the rectangle. Each

query uses $\mathcal{O}(\log_B \overline{N})$ I/Os, and as the number of neighbors is bounded by the constant t_N, the total to find all neighbors is $\mathcal{O}(\log_B \overline{N})$ I/Os

In the construction, it is needed to find the neighbors of some rectangle, e.g., when making a merge and to detect if the number of neighbors of the neighbors have increased beyond the neighbor bound, and when making horizontal splits.

For the following, consider a non-rightmost rectangle $r = [x^r, y^r[\times[v^r, w^r[$, i.e., $y^r \neq +\infty$. In order to find the right neighbors of r, it can be utilized that the rectangles are exclusive in the right and top coordinate, and thus the point (y^r, v^r) must be inside the bottom right neighbor. Let this rectangle be $r' = [x^{r'}, y^{r'}[\times[v^{r'}, w^{r'}[$, note that the left side of r' must be equal to y^r, i.e., $x^{r'} = y^r$. The rest of the right neighbors can then be found, using the same trick in the version axis, as the next neighbor must contain the point $(y^r, w^{r'})$. This can be repeated until the returned rectangle is the top right neighbor. Assume the last reported rectangle r' has top version $w^{r'}$. If the query for point $(x^r, w^{r'})$ returns rectangle r, we continue the search for the next right neighbor above r'.

Finding the left neighbor is however harder, as the left side of a rectangle is contained inside the rectangle. A point strictly left must therefore be queried. By having a B-tree over the values of the left sides of all rectangles, then it is sufficient to query the predecessor value of x^r among these values, as it must be inside the left neighbor, as no rectangle can exist in between. Note that the B-tree can contain more values, as long as it contains all left values of the rectangles, and that the B-tree only needs to be updated when performing a vertical split. Since the B-tree can only store values involved in the at most $\overline{N}$ updates, $\overline{N}$ is an upper bound on the size of the B-tree, and updates and predecessor queries on it can be be done with $\mathcal{O}(\log_B \overline{N})$ I/Os. In total, as the number of neighbors on either side is bounded by the constant t_N, the neighbors on either side can be found using $\mathcal{O}(\log_B \overline{N})$ I/Os.

3.3 Potential Functions

Updates to the persistent structure, when ignoring the potential cost of updating the partition of rectangles to restore the rectangle invariants, can be done with $\mathcal{O}(\log_B \overline{N})$ I/Os. Splitting and merging rectangles, in order to maintain the invariants, then uses further I/Os, which affect the update time. We amortize the cost of this rebalancing over the updates. In this section, we state the potential function Φ driving the analysis of our structure.

Horizontal and vertical splits are triggered by the rectangle having too many segments. Vertical splits are performed if there are many spanning segments and horizontal splits if there are many non-spanning segments. There must thus be a large potential when there are many spanning and non-spanning segments. The merge is triggered by an underflow in the number of spanning segments. The potential must therefore be large when there are few spanning segments. Finally, rectangles are split when there are too many neighbors on one side, so the potential must be large when there are many neighbors on the same side.

This leads to the following potential function Φ, which is the weighted sum of four potential functions, with S_r denoting the number of spanning segments in rectangle r, and I_r denoting the number of internal endpoints of segments in r, i.e., endpoints of segments, which are not equal to the bottom or top version of r. The constants c_O, c_U, c_I, and c_N are determined in Section 3.6.

$$\Phi = (c_O \cdot \Phi_O + c_U \cdot \Phi_U + c_I \cdot \Phi_I + c_N \cdot \Phi_N) \cdot R \cdot \log_B \overline{N}$$

The four subpotential functions are again defined as $\Phi_O = \sum_r \Phi_O^r$, $\Phi_U = \sum_r \Phi_U^r$, $\Phi_I = \sum_r \Phi_I^r$, and $\Phi_N = \sum_r \Phi_N^r$, where

$$\Phi_O^r = \max\{0, {}^{S_r}/_{R} - {}^1/_2\} \tag{1}$$

$$\Phi_U^r = \begin{cases} 0 & \text{if } y^r = +\infty \\ \max\{0, {}^1/_2 - {}^{S_r}/_{R}\} & \text{otherwise} \end{cases} \tag{2}$$

$$\Phi_I^r = \begin{cases} \max\{0, {}^{I_r}/_{R} - {}^1/_2\} & \text{if } y^r = +\infty \\ \max\{0, ({}^{I_r}/_{R} - {}^1/_2)\,(9 - 16 \cdot \min\{{}^{S_r}/_{R}, {}^1/_2\})\} & \text{otherwise} \end{cases} \tag{3}$$

$$\Phi_N^r = \max\left\{0, N_r^L - {}^{t_N}/_2\right\} + \max\left\{0, N_r^R - {}^{t_N}/_2\right\} \tag{4}$$

For a rectangle r, we denote the four potentials Φ_O^r, Φ_U^r, Φ_I^r and Φ_N^r for *overflow*, *underflow*, *internal endpoint* and *neighbor potentials*, respectively.

The max functions ensure that the potential is never negative. E.g., for spanning segments, subtracting ${}^R/_2$ from S_r insures that each rectangle does not have any potential for the first ${}^R/_2$ spanning segments, and then linear potential on the rest. Similarly for the rest of the potential functions.

The potential function Φ_I^r for internal points contains the same base function on the internal point count, but then depending on if the rectangle can contain underflow potential, this base function is multiplied by a function dependent on the amount of underflow. This leads to the value of the function being scaled larger when there is more underflow. As for all non-rightmost rectangles, it must hold that $S_r \geq R/4$, then the *scale value* $9 - 16 \cdot \min\{S_r/R, 1/2\}$ is in the interval $[1, 5]$.

Note that cloning and queries do not alter the potential. The potential must therefore be analyzed only for INSERT and DELETE updates. Any update can be split into two parts: updating a segment in some rectangle and a potential re-partitioning of the rectangles.

For each of the rebalancing operations, the colored predecessor structure and point location query structure need to be updated for the new rectangles. In Section 4 we describe that inserting a new rectangle into the colored predecessor structure requires amortized $\mathcal{O}(R \log_B \overline{N})$ I/Os, as each rectangle contains $\Theta(R)$ versions, and each version can be inserted in amortized $\mathcal{O}(\log_B \overline{N})$ I/Os. Further, we describe in Section 5 that inserting new rectangles into the point location query structure requires amortized $\mathcal{O}(R \log_B \overline{N})$ I/Os. As both operations must be performed with the released potential, then a rebalancing operation needs to release $R \log_B \overline{N}$ potential to pay for the work. In Section 3.5 and 3.6 we show that the factor preceding $R \log_B \overline{N}$ in Φ decreases by at least 1 for every rebalancing operation.

For each update operation without rebalancing, Φ may increase by no more than $\mathcal{O}(\log_B \overline{N})$ for updates to use amortized $\mathcal{O}(\log_B \overline{N})$ I/Os. We achieve this by showing in Section 3.4 that the factor preceding $R \log_B \overline{N}$ in Φ increases by $\mathcal{O}(1/R)$. Table 3 summarizes how the potential changes with and without rebalancing.

3.4 Update without Rebalancing

When performing update operations without rebalancing the rectangles, the partitioning of the rectangles remains unaltered, and thus the neighbors of all rectangles are unchanged. This results in $\Delta\Phi_N = 0$ for both update operations.

Insert. For an insert operation, a segment $\{x\} \times [v, \text{succ}(v)[$ is inserted into a rectangle $r = [x^r, y^r[\times [v^r, w^r[$. This may create a spanning segment if the rectangle only spans a single version, or the inserted value is inserted into the only version in the rectangle, where it was non-present, e.g., r already contains segments $\{x\} \times [v^r, v[$ and $\{x\} \times [\text{succ}(v), w^r[$. This leads to $\Delta\Phi_O \leq 1/R$. By creating a spanning segment, the underflow of spanning segments may decrease but not increase, resulting in $\Delta\Phi_U \leq 0$. The inserted segment may also create at most two new internal points in the rectangle. As the scale value is at most 5, then $\Delta\Phi_I \leq 10/R$.

Delete. For a delete operation, a segment $\{x\} \times [v, \text{succ}(v)[$ is removed from a rectangle $r = [x^r, y^r[\times [v^r, w^r[$. This can then break or delete a spanning segment, which increases the underflow of segments, leading to $\Delta\Phi_U \leq 1/R$. As the number of spanning segments can only decrease, then $\Delta\Phi_O \leq 0$. By breaking a segment, up to two new internal endpoints may be created in the rectangle. The two new internal points then increase the potential function by at most $10/R$. The decrease of spanning segments in the internal point potential function then leads to a further increase, since the scale value of Φ_I^r can increase. As the number of segments is at most $2R$, and at least $R/4$ are spanning, then the number of internal points is at most $7/2 \cdot R$, leading to an increase of at most $(7/2 - 1/2) \cdot 16/R = 48/R$. In total $\Delta\Phi_I \leq 58/R$.

3.5 Rebalancing Rectangles

If an update in rectangle r causes the invariants to be violated, we perform three phases of rebalancing operations to restore them. First, if the update is a deletion that causes the number of spanning segments to fall below $R/4$ then we perform a single merge to restore invariant I_2. The resulting rectangles contain up to $6R + 1$ segments as at most 3 rectangles are joined. Second, invariant I_1 is restored in all rectangles containing more than $2R$ segments by performing $\mathcal{O}(t_N)$ vertical and horizontal splits as described in Section 3.1. The first two phases may cause rectangles to have many neighbors. In phase three, we fix this by making neighbor splits until all rectangles have fewer than t_N neighbors, which restores invariant I_5. If the update is an insertion or deletion that causes the rectangle to contain more than $2R$ segments but not too few spanning segments, rebalancing starts from the second phase. In the remainder of this section, we show how the different subpotential functions change for every rebalancing operation.

Vertical split. We perform a vertical split only if the number of segments is $|S^r| > 2R$, and the number of spanning segments is $S_r \geq R$. The split is performed at the median value of the spanning segments.

This evenly partitions the spanning segments, such that both new rectangles contain $\geq R/2$ spanning segments. Thus, $\Delta\Phi_U = 0$. For Φ_O, due to the max function and that there now are two rectangles, the potential is released from the threshold in both the new rectangles, i.e., $\Delta\Phi_O = -1/2$.

For the internal points, there is no guarantee as to how they are distributed, but in any case, then $\Delta\Phi_I \leq 0$, since the number of internal points is unchanged.

When splitting the rectangle, all left neighbors become left neighbors of the left rectangle, and all right neighbors become right neighbors of the right rectangle, thus not altering the neighbor potential. Since $t_N \geq 2$, the two new rectangles being neighbors do not yield any increase in potential, and thus $\Delta\Phi_N = 0$.

Horizontal split. We perform a horizontal split only if the number of segments is $|S^r| > 2R$, and the number of spanning segments is $S_r < R$. Since each of the at least R non-spanning segments contributes at least one internal endpoint, then $I_r \geq R$. The median version among the internal endpoints is then found using the local versions, at which the split will occur. This evenly partitions the internal points into the new top and bottom rectangles, with some endpoints landing on the split line, which will no longer be internal points. If the rectangle has no underflow potential, a similar analysis as for vertical splits leads to $\Delta\Phi_I = -1/2$ and $\Delta\Phi_U = 0$.

If the rectangle has some underflow potential, then the scale value of the internal point potential function is larger than one. During the split, new spanning segments may be created, but worst-case no new spanning segments are created, and the underflow potential is transferred to both new rectangles. This results in $\Delta\Phi_U \leq 1/2 - S_r/R$ and $\Delta\Phi_I \leq -1/2\,(9 - 16^{S_r}/R)$. As the rectangle, in this case, has underflow potential, then it must be a non-rightmost rectangle, and thus $R/4 \leq S_r < R/2$.

When splitting horizontally, all spanning segments in the original rectangle become spanning in both new rectangles. Any other segment may become spanning in at most one of the two rectangles. In total, each segment may yield one extra spanning segment compared to before the split. As there are at most $6R + 1$ segments, and there is no potential for the first $R/2$ new spanning segments, then $\Delta\Phi_O \leq 5 + 1/2 + 1/R$.

Consider one side of neighbors. The split distributes these neighbors among the top and bottom rectangle, with at most one becoming the neighbor of both. None of the two new rectangles can have more neighbors than the original rectangle. If one rectangle has the same number of neighbors as the original rectangle, then the other has one neighbor. The neighbor potential thus does not increase internally among the rectangles. However, the split leads to the one neighboring rectangle at the split point, getting an additional neighbor. Considering both sides, this leads to $\Delta\Phi_N \leq 2$.

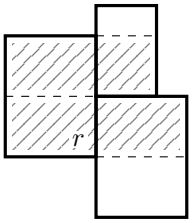
Neighbor split. A neighbor split occurs when the number of neighbors on one side of a rectangle is strictly larger than t_N, the number of segments is $\leq 2R$ and the number of spanning segments is $S_r \geq R/4$ if the rectangle is non-rightmost. The median version is selected among the versions where the neighbor changes and a horizontal split is made at this version. This distributes the neighbors evenly among the new top and bottom rectangle, thus decreasing the potential. However, for the other side, one split is made, leading to a potential increase of at most one neighbor. In total, this results in $\Delta\Phi_N \leq -t_N/2 + 1$.

By similar arguments as for horizontal splits, the at most $2R$ segments lead to $\Delta\Phi_O \leq 2R/R - 1/2 = 3/2$ and $\Delta\Phi_U \leq 1/2 - R/4/R = 1/4$. Note, however, that they cannot both increase. Since the internal points are distributed among the top and bottom rectangle and the scale value can only decrease, then $\Delta\Phi_I \leq 0$.

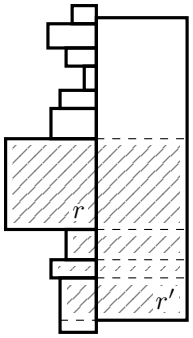
Merge. A merge occurs, immediately when a rectangle r is underflowing, i.e., when $R/4 - 1 \leq S_r < R/4$ and the rectangle is not a rightmost rectangle. Note that the number of segments in the rectangle is at most $2R + 1$ and that all neighbors that are non-rightmost rectangles contain at least $R/4$ spanning segments, and at most $2R$ segments. There are then multiple cases for how the merge is performed, to ensure not too many new neighbors are created. These cases can be seen in Figures 4 and 5.

Two neighbors on some side. This case is seen in Figure 5a. If one side of r contains exactly two neighbors, the neighbors on this side can be split, as well as r itself, to align the rectangles for joining. The three splits result in $\Delta\Phi_N \leq 3$. One half of the split neighbors are not joined, and using similar arguments as earlier, they cannot increase the potential of the internal points or underflow. The joined rectangles must contain at least $R/2$ spanning segments and thus have no underflow potential. This removes the underflow of r, resulting in $\Delta\Phi_U \leq -1/4$.

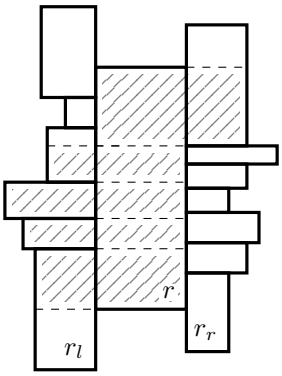
There are at most $R/2$ internal points in r with no internal endpoint potential, which now potentially get potential. However, as they are placed into rectangles, which contains at least $R/2 - 1$ spanning segments, the scale value is $1 + 16/R$. The potential on the already existing internal points may decrease, due to the increase in spanning segments and implied decrease in scale value, however, it cannot increase. Therefore $\Delta\Phi_I \leq 1/2 + 8/R$.



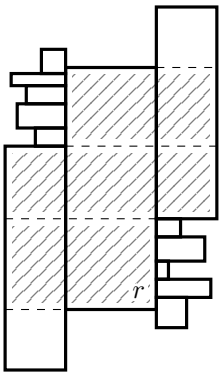
(a) Two neighbors to one side.



(b) One neighbor r' with underflow potential to one side.



(c) Multiple neighbors to both sides, with no overlap.



(d) Multiple neighbors to both sides, with overlap.

Figure 5: The different merge cases, with the underflowing rectangle being r, apart from when r is a leftmost rectangle. The dashed lines represent the location where the rectangles are horizontally split. The grayed areas indicate the newly joined rectangles.

The neighbor rectangles contain at most $2R$ spanning segments, and their overflow potential each increase by at most $3/2$. Rectangle r contains at most $2R+1$ segments. As these segments are distributed into two rectangles, each of which may contain $\geq R/2$ spanning segments, then all distributed segments may be spanning. The spanning segments of r, of which there are $< R/4$, may then become spanning in both new rectangles. The rest of the segments can only be spanning in one of the new rectangles. The extra $+1$ segment arrives from some old spanning segment that was broken and has a gap of at least one version. Therefore at most $2R - R/4$ of the remaining non-spanning segments may become new spanning segments. This results in $\Delta\Phi_O \leq 5 + 1/4$.

Note that this case is a special case of the basic merge seen in Figure 4.

One neighbor on some side. This case is shown in Figure 5b. If some side has one neighbor r', and the other does not have two neighbors, then it is favorable to join with r', as this limits the neighbor increase in r'. To join r with r', r' may need to be split twice (at the top and bottom of r), leaving two additional rectangles with the same underflow as r'. If r' has no underflow, this is not a problem, and the merge can stop by simply joining r into the split middle section of r'. For the following, assume that r' has underflow potential.

In this case, one of the two created rectangles from r' must have its underflow potential removed to release the potential needed for the operation. Consider the same side of r' as the side r is on. This side must contain some neighbors above and below r. Choose the half, which contains the least number of neighbors, and split the part of r' to this half, and join all of the rectangles, which then must remove the underflow. It must hold that this half has at most $t_N/2$ neighbors, leading to at most $t_N/2 + 2$ splits when factoring in the extra splits to align the neighbors with r' and the split from r. By a similar argument as above, it must hold that $\Delta\Phi_U \leq -1/4$, as the underflow potential of r is released.

The at most $2R$ segments of r' are split into at most $t_N/2 + 2$ rectangles, where they may be new spanning segments in all but one of them, increasing the overflow potential by $\leq t_N + 2$. Rectangle r was underflowing and had less than $R/4$ spanning segments, and therefore the first $R/4$ spanning segments added to r from r' do not increase the potential. The last split rectangle increases the potential by $3/2$ by a similar argument as above. In total resulting in $\Delta\Phi_O \leq t_N + 2 - 1/4 + 3/2 = t_N + 3 + 1/4$.

Consider the internal points of r', where some are added to r and the rectangles below or above r. By a similar argument as above, it also holds that $\Delta\Phi_I \leq 1/2 + 8/R$.

Assume that r' has k neighbors on the side r is on, and only consider the change in the neighbor potential for these rectangles. Then at most $\lceil k/2 \rceil$ neighbors will change from being a neighbor of r' into being a neighbor of other rectangles. The rounding is when k is odd, as there may be $\lfloor k/2 \rfloor$ neighbors on either side of r, and r is one of the rectangles which no longer is a neighbor of r'. Worst case, they all become neighbors of the same rectangle, increasing the neighbor potential in this rectangle by at most $\lceil k/2 \rceil$. The difference in potential of r' is at most $\max\{0, \lfloor k/2 \rfloor - t_N/2\} - \max\{0, k - t_N/2\}$. Lastly, an aligning split increases the neighbor potential by at most 1. In total, the potential increase is maximized for $k = t_N/2$ resulting in $\Delta\Phi_N \leq t_N/4 + 2$.

Note that this case uses the basic merge seen in Figure 4 twice, once for merging r into r', and once to merge the top or bottom rectangle of r' into its neighbors.

Multiple neighbors on both sides. In this case, many splits must be made. If r is joined with the neighbors to the left, this may increase the number of neighbors for the rectangles on the right side. If, however, all of these right neighbors only originally have r as a left neighbor, then the neighboring potential for these rectangles cannot increase in total, as the neighbor potential r has on the right is released.

This holds for all right neighbors, except for the top and bottom. If r is split to align with these, it can then freely be split in between. To not increase the number of neighbors of the top and bottom rectangle on the left, the splits are made to align with the side, which covers most of r. This can be seen in Figure 5c, where r is split at the top of r_l instead of at the top of r_r, as r_l covers most of r. The top of both rectangles cannot however be directly compared, apart from checking if they are identical, but by making a point query at (x^{r_r}, w^{r_l}), i.e., at the top value of r_l, but at the left side of r_r, then this point will be inside r_r only if the top of r_l is below the top of r_r. Similarly for the top neighbors.

The selected top and bottom neighbors may overlap with the part of r they cover, as seen in Figure 5d, further creating two cases. This happens only when the selected neighbors are from each side. If they do not overlap but exactly cover r, then this reduces to the case with two neighbors. Note that overlapping can be checked with a point query, like above. For both cases, it holds that $\Delta\Phi_U \leq -1/4$, as the underflow potential of r is removed, and no new underflow potential is introduced, and $\Delta\Phi_I \leq 1/2 + 8/R$ by similar arguments as above, as the internal points of r are distributed into rectangles with no underflow potential.

No overlap. This is the case illustrated in Figure 5c.In this case, four splits are initially made, two on the selected neighbors and two in r, and potentially a fifth split to align the space between the top and bottom split. All joined rectangles in between become neighbors of rectangles that previously only had r as a neighbor, therefore not increasing the potential. In total, the neighbor potential may only increase for the 3 splits outside of r, resulting in $\Delta\Phi_N \leq 3$. Rectangle r is split at most $t_N - 1$ times. The increase in overflow potential for the other three split rectangles is at most $3/2$ each, by a similar argument as above. In r there are at most $R/4$ spanning segments, which may be spanning at most t_N rectangles, and the remaining segments may be spanning in at most $t_N - 1$ rectangles, by similar argument as above. In total resulting in $\Delta\Phi_O \leq 2t_N + 3 + 3/4$.

Overlap. This is the case illustrated in Figure 5d. In this case, the selected neighbors overlap vertically. By further splitting these neighbors, r can be joined into three rectangles. The splits in the neighbors result in $\Delta\Phi_N \leq 4$. By similar arguments as above, the increase in overflow potential for r is at most $4 + 1/4$. In the neighboring rectangle, there are at most $2R$ segments, which all may become spanning in two new rectangles. Each of the three rectangles does not have potential for the first $R/2$ spanning segments, resulting in total $\Delta\Phi_O \leq 10 + 3/4$.

Leftmost rectangles. Rectangles at the rightmost side do not have underflow potential and thus cannot trigger a merge. Rectangles at the leftmost side, however, only have neighbors on the right side. In this case, r must be merged with the right neighbors, however, as it has no left neighbors, the neighbor potential on this side cannot increase. This is done as seen in Figure 4. By similar arguments as earlier, it can be shown that $\Delta\Phi_U \leq -1/4$, $\Delta\Phi_I \leq 1/2 + 8/R$, $\Delta\Phi_N \leq 2$ and $\Delta\Phi_O \leq 2t_N + 1 + 1/4$.

Table 3: Difference in the subpotential functions for the different update operations and rebalancing operations.

Operation	$\Delta\Phi_N$	$\Delta\Phi_O$	$\Delta\Phi_U$	$\Delta\Phi_I$
Insert	0	$\leq 1/R$	≤ 0	$\leq 10/R$
Delete	0	≤ 0	$\leq 1/R$	$\leq 58/R$
Vertical split	0	$-1/2$	0	≤ 0
Horizontal split ($\Phi_U^r = 0$)	≤ 2	$\leq 5 + 1/2 + 1/R$	0	$-1/2$
($\Phi_U^r > 0$)	≤ 2	$\leq 5 + 1/2 + 1/R$	$\leq 1/2 - S_r/R$	$\leq -1/2 \cdot (9 - 16 S_r/R)$
Neighbor split	$\leq -t_N/2 + 1$	$\leq 3/2$	$\leq 1/4$	≤ 0
Merge				
(Fig. 5a, $N_r^L = 2 \vee N_r^R = 2$)	≤ 3	$\leq 5 + 1/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Fig. 5b, $N_r^L = 1 \vee N_r^R = 1$)	$\leq t_N/4 + 2$	$\leq t_N + 3 + 1/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Fig. 5c, $N_r^L \geq 3 \wedge N_r^R \geq 3$, no overlap)	≤ 3	$\leq 2t_N + 3 + 3/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Fig. 5d, $N_r^L \geq 3 \wedge N_r^R \geq 3$, overlap)	≤ 4	$\leq 10 + 3/4$	$\leq -1/4$	$\leq 1/2 + 8/R$
(Fig. 4, $x^r = -\infty \wedge N_r^R \geq 3$)	≤ 2	$\leq 2t_N + 1 + 1/4$	$\leq -1/4$	$\leq 1/2 + 8/R$

3.6 Potential Function Constants

The differences in the subpotential functions, as calculated in Section 3.4 and Section 3.5 can be seen summarized in Table 3. For the insert and delete rows at the top, the differences are without rebalancing.

In Φ the subpotential functions are a linear combination. In the table, each rebalancing operation row must decrease by 1 to release $R \log_B \overline{N}$ potential. In order to find working constants c_N, c_O, c_U and c_I, the rows of the table can be viewed as constraint for a linear program, where each row must be less than -1. Some values are dependent on t_N, which is a constant. In order to find a solution, the linear program is tried solved for increasing values of t_N, until a solution is found. Some rows are dependent on R, with R appearing below the fraction line. By assuming that $R \geq c$, for some constant c, then all greater values of R must yield smaller added values, and therefore the found constants are also a valid solution to these smaller values. For smaller values of R, then the geometric structure is not used, but the updates stored in a list, and all operations work by scanning or appending to this list. Note that $R \geq c \Leftrightarrow \overline{N} \geq \sqrt[B]{B}^c$, and as $\sqrt[B]{B} \leq \sqrt[e]{e}$, then this results in $\overline{N}$ being smaller than a constant, resulting in all operations requiring $\mathcal{O}(1)$ I/Os, which therefore satisfies the desired bounds. Choosing larger values of c yields to smaller values of t_N. For the below displayed solution, c is chosen to be 32 for the merge cases. To get integer outputs, $R \geq 2$ is used for the horizontal split cases, which only leads to negligible larger constants.

For the constraint from horizontal split with underflow potential, $\Delta\Phi_U$ and $\Delta\Phi_I$ are not constants since the scale value can change. However, they are linear bounds in S_r . By creating two constrains for the minimum and maximum value of S_r, which is $R/4$ and $R/2$ respectively as there must be underflow potential in this case, the program may be solved.

The objective function to minimize in the linear program, is chosen to be the sum of the constants. This yields the following constants

$$t_N = 34 \quad c_N = 822 \quad c_O = 2 \quad c_U = 52592 \quad c_I = 5922 \,.$$

These constants can then be used to check if the constraint from horizontal split with underflow potential is satisfied, by calculating the left side of the constraint:

$$822 \cdot 2 + 2 \cdot 6 + 52592 \cdot (1/2 - S_r/R) + 5922 \cdot (-1/2 \cdot (9 - 16 S_r/R)) = 1303 - 5216 S_r/R \,.$$

This value is large, when S_r is small. The smallest value of S_r is $R/4$, resulting in the maximum value of the left side being -1. The constraint is therefore satisfied, and the solution to the constants is valid.

3.7 Maintaining Structure inside a Rectangle

The remaining invariants to maintain are I_3 and I_4. For update operations without rebalancing, the updated segments can be inserted into S^r to maintain the order. An update may add new versions to r, and using the colored predecessor structure, the new versions can be inserted into L^r at the right position. An update may also delete the last endpoint of versions, which then must be deleted from L^r, which

can be determined by a scan of the segments. After updating L^r, the segments can then be scanned, to update the local versions of the endpoints.

For the rebalancing operations, first note that S^r can maintain the value order during horizontal and vertical splits, as the segments may be partitioned in order, and for vertical joins all segments in one rectangle are before all rectangles in the other, and the lists can then simply be concatenated.

For a horizontal split L^r is split at some version, and the resulting segments in the upper rectangle can be scanned to update the local versions. For a vertical split, L^{r_1} and L^{r_2} must be reconstructed from the endpoints of the segments in the rectangles. As binary merge-sort on $\mathcal{O}(R)$ elements uses $\mathcal{O}(R/B \log_2 R) = \mathcal{O}(R \log_B \overline{N})$ I/Os, then $\mathcal{O}(R)$ elements can be sorted a constant number of times using the released potential. This allow for constructing L^{r_1} and L^{r_2} for the resulting rectangles from the endpoints of the segments, by sorting them by the local version in L^r.

When vertically joining two rectangles, two version lists L^{r_1} and L^{r_2} needs to be merged into L^r. Firstly, the start and end of the two list must be equal, as the top and bottom version of the rectangles are equal. Using the colored predecessor structure, before it is updated, the versions may be merged, as the predecessor query can be used to determine the ordering. As the list contains $\mathcal{O}(R)$ versions, and each predecessor uses $\mathcal{O}(\log_B \overline{N})$ I/Os, the resulting $\mathcal{O}(R \log_B \overline{N})$ I/Os can be payed by the released potential. The local ordering on the segments can then be done by a constant number of sorts and scans.

3.8 Space Usage

Each update operation increases the value of the subpotential functions by $\mathcal{O}(1/R)$. Each rebalancing operation decreases the subpotential functions by at least 1, and creates $\mathcal{O}(1)$ new rectangles. There are at most $\overline{N}$ updates performed on the structure, resulting in $\mathcal{O}(\overline{N}/R)$ rectangles. Each rectangle uses $\mathcal{O}(R/B)$ blocks of space, i.e., the total space usage is $\mathcal{O}(\overline{N}/R) \cdot \mathcal{O}(R/B) = \mathcal{O}(\overline{N}/B)$ blocks.

4 Colored Predecessor Queries

In this section, we present our solution to the colored predecessor problem. That is, given a version v and a rectangle r, find the predecessor version of v in the local version list L^r. We consider each rectangle to represent a unique *color*. The global version list L is ordered from left-to-right such that the predecessor of a version is to the left of it. We overload the notation of r such that it also defines the color corresponding to the rectangle r and let $C = \{1, 2, 3, \ldots\}$ be the set of colors. In the point location structure in Section 5 we use the special case with only a single color ($|C| = 1$) to find the predecessor of a version among the bottom versions of the rectangles. The following theorem states our result for the colored predecessor problem.

Theorem 3. *Let N denote the total number of updates to the global version list L and all local version lists L^r. Then there exists a data structure that given a version $v \in L$ and a color r, can find the predecessor u of v in L^r in worst-case $\mathcal{O}(\log_B N)$ I/Os. The structure supports insertions of versions in both L and L^r and deletions from L^r in amortized $\mathcal{O}(\log_B N)$ I/Os. The space usage is linear in the total number of updates.*

We store all versions from the global version list L and all local version lists L^r at the leaves of a B-tree ordered by the global version list, i.e., the same version can appear multiple times. For every local version $v_r \in L^r$, we conceptually color the path from the leaf containing v_r to the root by the color r, i.e., an internal node can have up to $|C|$ colors. The predecessor of a version v among the versions in L^r can now be found by following the path from v towards the root until a node of color r is found that has child of color r that is to the left of the search path, from which the search reverses towards the leaves following the rightmost nodes of color r (starting with a child to the left of the path). See Figure 6a for an example. To avoid storing a color at every node on the path to the root we observe that when considering the colored paths from the root towards the leaves it suffices to only color the nodes where paths branch. The number of branches is $\mathcal{O}(N)$. To achieve query complexity $\mathcal{O}(\log_B N)$ we use the idea of *down pointers* [28] to avoid searching in every node among its colors. Finally, we make the structure dynamic using a weight-balanced B-tree [5].

4.1 Static Predecessor Queries

We first give a simple structure for the colored predecessor problem for the static case, capturing the basic idea of our construction.

As mentioned, we store all versions from the global version list L and all local version lists L^r at the leaves of a B-tree. The versions from L appear in their left-to-right order, and each version $v \in L$ is preceded by all its copies in arbitrary color order. The resulting list has size $|L| + \sum_{r \in C} |L^r| = N$. The B-tree has height $\mathcal{O}(\log_B N)$. Finally, for every copy v_r we color all the nodes on the leaf-to-root path from v_r with the color r. A node can be colored with multiple colors. We let C_u denote the colors stored at a node u, which are exactly the colors appearing at the leaves in the subtree rooted at u. To perform the search efficiently, for each $r \in C_u$, we store the list of children of u colored r in order.

If we at each node store C_u as a B-tree (ordering the colors by their identifier), it takes $\mathcal{O}(\log_B |C_u|)$ I/Os to find the relevant color $r \in C_u$. However, as every internal node may have $|C|$ colors causing a colored predecessor search to take worst-case $\Theta(\log_B N \cdot \log_B |C|)$ I/Os. The space usage is $\mathcal{O}((N/B) \log_B N)$, since for each version $v \in L^r$ we store color r at all ancestors of the leaf node v_r, i.e., the space usage is super linear.

4.2 Improving Query Complexity

We first describe how to improve the query complexity using a variation of *downpointers* [28], which is a simple case of fractional cascading [15]. To avoid searching in every node for the color r we augment each node u with a secondary structure C_u (we overload notation and let C_u denote both the set of colors stored at node u and the secondary structure at u). Now C_u is represented as a blocked linked list where for every color r of u it stores pointers $C_{u,r}$ directly to each of the at most B children of u that are also colored by r. We call $C_{u,r}$ a *chunk* and these pointers are stored consecutively in the linked list in $\mathcal{O}(1)$ blocks. In fact, if u has $k \leq B$ children that are colored by r then the pointers in $C_{u,r}$ directly point to the chunk $C_{u_i,r}$ in C_{u_i} for $1 \leq i \leq k$. For a specific color r, we call these *r-downpointers*. We also have *reverse* r-downpointers, by letting each chunk $C_{u_i,r}$ have a pointer to $C_{u,r}$. At the root $\hat{u}$ every color is represented in $C_{\hat{u}}$. We keep a separate B-tree over the colors at the root where every color r in the B-tree has a pointer to $C_{\hat{u},r}$.

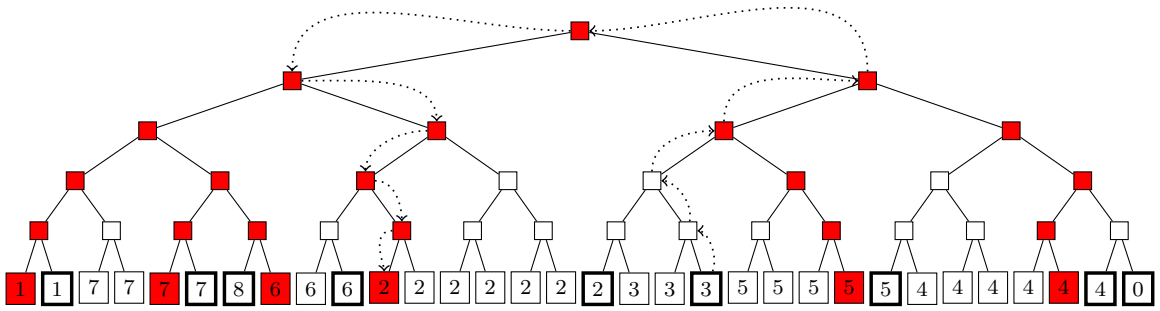
Inspired by the previous approach, the predecessor query (v, r) follows a simple path p from the leaf containing $v \in L$ towards the root $\hat{u}$, this time always ending up at the root $\hat{u}$. Using the B-tree at $\hat{u}$ we then find the chunk $C_{\hat{u},r}$, and follow the nodes colored by r using the r-downpointers for as long as they overlap with the simple path p. When the paths diverge, that is the next node on p is not colored by r, we similarly to before choose the child immediately to the left that is also colored by r, and follow all the rightmost nodes colored by r still using the r-downpointers. If no left sibling colored r exists at the diverging node, we follow the path back towards the root and begin from the first such node with a left sibling colored r.

This change improves the I/Os required for predecessor queries to $\mathcal{O}(\log_B N)$ since we spend $\mathcal{O}(1)$ I/Os for every node on the path to the root and down, except for the root where we spend $\mathcal{O}(\log_B N)$ I/Os.

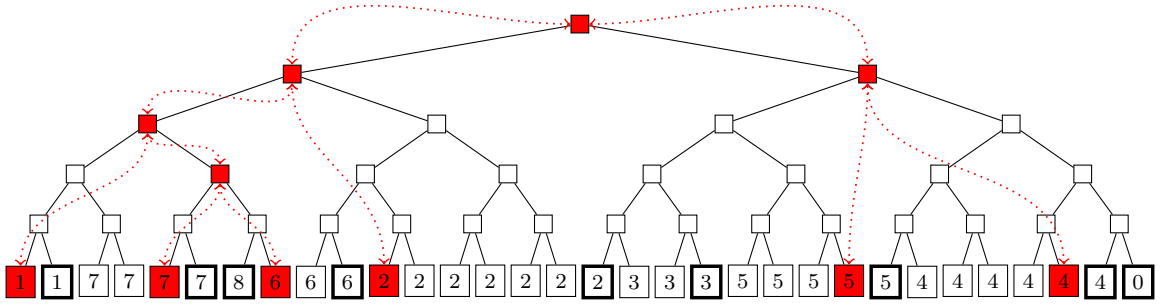
4.3 Improving Space Complexity

The factor $\mathcal{O}(\log_B N)$ space overhead was caused by all nodes being colored on leaf-to-root paths. To remove it, for every color r we only color the nodes u with multiple outgoing r-downpointers. We call these nodes *r-branches* and observe that at most $|L^r| - 1$ such nodes exist for every color. The r-downpointers now directly jump between r-branches in the tree. See Figure 6b for an example. To still have an ordering between r-downpointers at a node they also store which child of u they point in the direction of.

Performing queries is now slightly different. For a predecessor query (v, r) we again follow the simple path from the leaf $v \in L$ to the root $\hat{u}$ and use the B-tree at $\hat{u}$ to find the chunk $C_{\hat{u},r}$. This gives us access to the r-downpointers and we follow them until as long as they overlap with the simple path p. To follow the r-downpointers in the direction of p we use their ordering. The r-downpointers may now skip many nodes in the tree and we find the last r-branch u overlapping with the path p together with the r-branch u_c immediately after u following the r-downpointers. The node u_c does not overlap with the path p but we follow the simple path from u_c towards u until we find a node u_p that overlaps with p. If the path from u_c to u_p is to the left of the path from v to u_p then we find the predecessor of v by always following the rightmost r-downpointer starting from the node u_c. Here we again use the ordering between r-downpointers. If the path from u_c to u_p is to the right of the path from v to u_p then we follow the path of r-branches from u towards the root using r-downpointers until we find an r-branch with at least one r-downpointer going left of the path. Among these r-downpointers we follow the rightmost one and continue to do so towards the leaves to find the predecessor as before. The number of I/Os required to perform queries remains $\mathcal{O}(\log_B N)$ since we traverse a constant number of paths of length at most



(a) The simple structure where the dotted path indicates the search for the predecessor of version 3 among the red versions. Note here the first node encountered with a left child (not on the path to the root) that is red is the root itself.



(b) The efficient structure with the red branches, downpointers and reverse downpointers shown.

Figure 6: Our simple (6a) and efficient (6b) colored predecessor structure which for clarity is a binary tree instead of a B-tree and with only one color shown. At the leaves of the tree we have the global version list $1, 7, 8, 6, 2, 3, 5, 4, 0$ (same as in Figure 1) indicated by the leaves in bold. Preceding each version from the global version list we have its copies in arbitrary order.

the height of the tree and spend $\mathcal{O}(1)$ I/Os for every node due to r-downpointers. The space usage is now $\mathcal{O}(N/B)$ since the total number of r-downpointers and reverse r-downpointers stored in the tree is $\mathcal{O}\big(\sum_{r \in C} |L^r|\big) = O(N)$.

4.4 Dynamic Colored Predecessor Problem

In this section, we describe how to make the structure dynamic, since updates in the geometric view may trigger insertions in the global version list L and insertions and deletions in L^r. To keep the B-tree balanced under updates nodes are split (we avoid merges by never explicitly performing deletions) and secondary structures need to be rebuilt. Thus, we replace the standard B-tree with a weight-balanced B-tree [5].

We first consider updates that do not trigger the splitting of nodes in the weight-balanced B-tree. The most straightforward update is when a version v is deleted from L^r. We handle this by leaving v_r in the B-tree as a dead uncolored leaf, and thus no node splits are triggered. To uncolor v_r we identify the single affected r-branch using the reverse r-downpointers from v_r and update the affected r-downpointers using amortized $\mathcal{O}(1)$ I/Os. Note the amortization is due to deletions in the secondary structure C_u which is represented as a blocked linked list where deletions may cause blocks to underflow and have to be merged. Similarly, for insertions, a block may have to be split. In both cases, all downpointers moved to a new block must also update the corresponding reverse downpointer to point to the new block. Both cases require updating $\Theta(B)$ downpointers which can be done in amortized $\mathcal{O}(1)$ I/Os similar to normal array doubling.

Inserting a new version v in the global version list L inserts v in the B-tree right after the predecessor of v in L. We perform no updates of secondary structures unless nodes split.

Similarly, a colored version v_r is inserted in the B-tree just before v in L. We then need to conceptually color the path from v_r towards the root $\hat{u}$, which may introduce a new r-branch. Let v_{pred} be the

predecessor of v_r among L^r and similarly let v_{succ} be the successor. The simple path p from v_r to $\hat{u}$ is surrounded by the corresponding simple paths from v_{pred} to $\hat{u}$ and v_{succ} to $\hat{u}$. The potential new r-branch then exists at an intersection along these three paths. The cases where v_{succ} or v_{pred} does not exist are similar. We use the existing structure to find v_{pred} and v_{succ}. Note that finding the successor is symmetric to finding the predecessor. We then follow the three paths synchronously until the path p collides with one (or both) of the other paths. The node u where they collide potentially defines a new r-branch, and we update the surrounding r-downpointers. Since the r-downpointers are located together in a chunk and at most 4 nodes have to update their r-downpointers we do this using amortized $\mathcal{O}(1)$ I/Os plus the $\mathcal{O}(\log_B N)$ I/Os required to traverse the paths from the leaves to the root $\hat{u}$. This concludes how we handle updates when no nodes in the B-tree are split.

4.5 How to Split Nodes

We now consider insertions that trigger node splits in the weight-balanced B-tree and require secondary structures to be rebuilt. The weight $w(u)$ of a node u is defined as the number of elements stored in the leaves of the subtree rooted at u. The main property we will use is Lemma 3.7 of [5] which states that whenever a node is split then $\Omega(w(u))$ insertions must have occurred in the subtree rooted at u since u was last split. Thus, since we allow insertions to take amortized $\mathcal{O}(\log_B N)$ I/Os, then we can free $\Omega(w(u))$ potential when u is split to rebuild the secondary structures. Crucially, $|C_u| = \mathcal{O}(w(u))$ since each version in the leaves contributes at most one downpointer and the number of reverse downpointers is bounded by the number of downpointers.

Now splitting node u into two new nodes u_l and u_r only affects downpointers and reverse downpointers that point from, to, or in the direction of u. The downpointers from u can be distributed to C_{u_l} or C_{u_r} using $\Theta(w(u)/B)$ I/Os, since we store with every r-downpointer its direction (the first node it passes through). Note if there is a color r where $C_{u,r}$ contains r-downpointers towards both the children of u_l and the children of u_r, then at least one r-downpointer is inserted in the parent of u. Furthermore, if there is only a single r-downpointer towards the children of u_l (similarly for u_r), then u_l is not an r-branch, and the r-downpointer is removed from u_l and instead inserted in the parent of u. The total number of r-downpointers inserted in the parent of u is bounded by $w(u)$ and can be inserted using amortized $\mathcal{O}(w(u))$ I/Os. Updating the reverse downpointers from u requires $\mathcal{O}(w(u))$ I/Os since each r-downpointer inserted in the parent of u causes $\mathcal{O}(1)$ related reverse r-downpointer to be updated or created. When we update the r-downpointers and reverse r-downpointers from u we also update the corresponding r-downpointers and reverse r-downpointers that previously pointed to u using amortized $\mathcal{O}(w(u))$ I/Os. Finally, the $\mathcal{O}(B \cdot w(u))$ r-downpointers from the parent of u in the direction of u must be updated to point in the direction of either u_l or u_r. We do this by marking the node pointed to by each r-downpointer. Then we traverse the subtree rooted at each child of u that will also be a child of u_l (similarly for u_r) and for every marked node encountered we update the corresponding r-downpointer to point in the direction of u_l. Traversing all the subtrees take $\mathcal{O}(w(u)/B)$ I/Os and marking and updating the direction of the r-downpointers takes $\mathcal{O}(w(u))$ I/Os.

Thus, nodes in the weight-balanced B-tree can be split using amortized $\mathcal{O}(w(u))$ I/Os which concludes the proof of theorem 3.

4.6 Combining with the Geometric Structure

We now briefly describe how our colored predecessor structure combines with the geometric construction from Sections 2 and 3 based on the following corollary:

Corollary 1. *Let $\overline{N}$ be the parameter from Theorem 1 giving a bound on the number of updates to the fully persistent search tree and let N be the number of updates to the colored predecessor structure from Theorem 3 triggered by these updates. Then $N = \mathcal{O}(\overline{N})$.*

Proof. Each of the at most $\overline{N}$ updates (Clone, Insert or Delete) introduces $\mathcal{O}(1)$ new versions into either the global version list L or into some local version list L^r, when no new rectangles are created. When a new rectangle is created then $\mathcal{O}(R)$ versions must be inserted into this new local version list. From Section 3.8 we know that at most $\mathcal{O}(\overline{N}/R)$ new rectangles are created and thus the number of updates N to L and L^r is $\mathcal{O}(\overline{N})$. □

Thus, when using the data structure for the colored predecessor problem described in Theorem 3 we can replace N by $\overline{N}$ in all the bounds as required for our rectangular partitioning based on the discussion in section 2.4.

5 Point Location

In this section, we present our solution on how to find the unique rectangle containing a query point (v, x), where v is a version and x is a value. Recall that we consider a disjoint rectangular partition of the plane as described in Section 2.3. At first, this appears to be an orthogonal planar point location problem. However, a crucial difference in our setting is that versions (corresponding to the vertical axis) are ordered according to their position in the version list. Because of this, it is hard to simply apply existing algorithms, and in our approach we will only ever compare versions for equality. That is, we never test if $v_i < v_j$ which naively would require the order-maintenance data structure by Dietz and Sleator [18] where order-queries use $\Theta(1)$ I/Os. Instead, we formulate it as a *dynamic ray shooting* problem on the bottom segments of each rectangle. Throughout this section, we assume that given a version v we can quickly find the predecessor of v among the versions represented by bottom segments. This is a special case of the colored predecessor problem we solve in Section 4 and allows us to assume that v is a version corresponding to a bottom segment of some rectangle and that for insertions of new segments, we know its correct position among the existing segments, with regards to the global ordering of versions. The bounds we get are summarized in Theorem 4.

Theorem 4. *There exists a semi-dynamic ray shooting data structure for non-overlapping horizontal segments, supporting ray shooting queries in worst-case $\mathcal{O}(\log_B S)$ I/Os and insertions in amortized $\mathcal{O}(B \log_B^2 S)$ I/Os, where S is the number of segments inserted. The space usage is $\mathcal{O}(S \log_B S)$ blocks. Furthermore, segments are only compared on the vertical axis for equality, that is, testing if they are at the same height.*

For our purposes $S = \mathcal{O}(\overline{N}/R)$, since that is the number of rebalancing operations on rectangles as described Section 3.8, which is what causes insertions in the point location structure. This results in the promised bound for point location queries in Section 2.4, and the insertions are within the potential released as described in Section 3.3. Note that the space usage is $\mathcal{O}(\overline{N}/B)$ blocks.

To see how this theorem applies to our problem we first observe that it suffices to consider insertions in the structure. Each segment corresponds to the bottom of a rectangle and thus the primitive rectangle transformations (Figure 3) and consequently the merge rectangle transformations (Figures 4 and 5) define how the set of segments change. Crucially, in all cases any point covered by a segment must still be covered, i.e., the area covered by segments only increases. Thus, using only insertions we can make sure that a given ray shooting query finds some segment at the correct version, by only inserting the segments that cover previously uncovered areas. For every version, we maintain the real set of segments created by rectangle transformations in a B-tree, sorted on the horizontal axis. Since there are at most S segments we can now find the correct segment to report with an overhead of only $\mathcal{O}(\log_B S)$. The overhead for insertions is similar.

Our approach is to store the segments in a segment tree T. For every node u in T, we have a secondary structure with a subset of the versions corresponding (primarily) to segments that span the horizontal interval defined by u. For a query (v, x) we find the predecessor of v in all the secondary structures on the path down to the leaf in T which contains x in its interval, as one of these predecessors will be the segment we are looking for. To determine the real predecessor among the candidate predecessors from the path we perform consecutive predecessor queries efficiently using a variation of fractional cascading [15]. Finally, since we need a dynamic version of the structure we use a weight-balanced B-tree [5].

5.1 Static Ray Shooting

We model the problem as two-dimensional vertical ray shooting among S horizontal non-overlapping segments where the i'th segment has the form (v_i, x_i, y_i). Here x_i and y_i represent the interval $[x_i, y_i[$ on the horizontal axis and v_i the position of the segment on the vertical axis. As mentioned, we only compare versions for equality, i.e. testing if $v_i = v_j$ by their version identifiers.

5.1.1 The Segment Tree

We first create a segment tree T on the elementary intervals on the horizontal axis defined by the endpoints of all segments. The number of elementary intervals is at most $2S + 1$ and we store them in a B-tree of degree $\delta = B/2$. That is every internal node except the root has degree $[\delta/2, \delta]$ and leaves store between $B/2$ and B elementary intervals. Thus, its height $T_h = \mathcal{O}(\log_B S)$ and the number of nodes of T is $|T| = \mathcal{O}(S/B)$. The segments are now stored in secondary structures in T. In particular, at every node

u and every elementary interval in a leaf, we have a secondary structure. Observe that each secondary structure is uniquely defined by an interval. As in a typical segment tree, we have the following properties:

P1 An internal node u corresponds to the interval $Int(u)$, that is the union of all elementary intervals stored in the leaves in the subtree rooted at u. Equivalently, the interval is the union of the intervals of the children of u.

P2 Each segment is stored in the secondary structure at node u (similarly for elementary intervals) if the segment spans the interval of u, but not the interval of $parent(u)$. That is segments are stored as high up the tree as possible in secondary structures which have non-overlapping intervals that exactly cover the segment.

P3 By property **P2** then for any value x then every segment spanning x will be stored in one of the secondary structures on the path from the root of T to the elementary interval containing x.

5.1.2 The Secondary Structure

We have a secondary structure D_u for every node u (similarly for elementary intervals) and thus the total number of secondary structures is $\mathcal{O}(S)$. The secondary structure D_u is defined by a unique interval $Int(u)$ and consists of a subset of the set of versions defined by the segments. We represent D_u as a blocked linked list in sorted order where each block contains B consecutive versions. In addition to the typical properties for segment trees, D_u satisfies the following:

P4 For any child u_c of u then the versions in D_{u_c} are also stored in D_u. That is, versions propagate up in the secondary structures.

P5 Every version in D_u that is stored due to property **P2** is called a *spanning* version. Versions that are not spanning contain a pointer to their predecessor among the spanning versions in D_u.

P6 The first version in each block in D_u (every B'th version) is also stored in D_{u_c} for every child u_c of u. These special versions we call *bridges* and they contain a bidirectional pointer between D_u and D_{u_c}. Note that this may cascade.

We are now ready to describe the algorithm for ray shooting precisely. In the following let (v, x) be the version v and value x which is the starting point of the ray. Let T be a segment tree as described above and let $\hat{u}$ be the root of T. For any node or elementary interval u of T let D_u be the secondary structure at u as described above and let *isElementaryInt*(u) be a function returning a boolean indicating if u is an elementary interval. The function *spanning_pred*(v, D_u) returns the predecessor of v among the spanning versions in D_u as described by property **P5**. The function *child*(x, u) returns the child u_c of u such that $x \in Int(u_c)$. The function *pred*(v, D_{u_c}) returns the predecessor of v in D_{u_c}. The function *findAbove*(v_{u_c}, D_u) returns a pointer to the version v_{u_c} in D_u, which exists by property **P4**.

We then perform the ray shooting by querying all the secondary structures on the path from the root $\hat{u}$ to the elementary interval containing x, and in each structure finding the predecessor of v among the spanning versions. The pseudocode is shown in Figure 7. An illustration of the RayShooting algorithm run on the secondary structure for three consecutive levels of the tree, can be seen in Figure 8.

```
RayShooting(v, x, u)
1  if isElementaryInt(u) then // Base of recursion
2      return spanning_pred(v,D_u)
3  u_c = child(x, u)
4  v_{u_c} = RayShooting(pred(v, D_{u_c}), x, u_c)
5  v_u = findAbove(v_{u_c}, D_u)
6  if spanning_pred(v_u, D_u) = spanning_pred(v, D_u) then
7      return v_u
8  else
9      return spanning_pred(v, D_u)
```

Figure 7: The RayShooting algorithm.

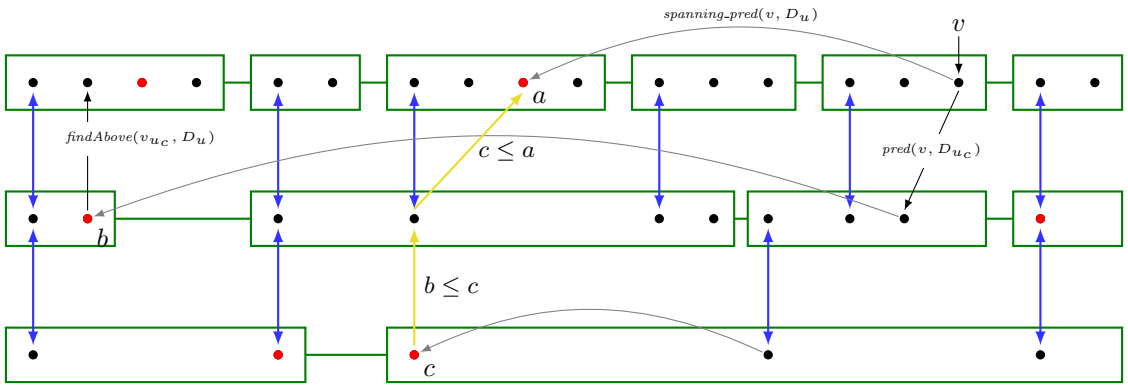


Figure 8: The RAYSHOOTING algorithm run on the secondary structure, with input version v, value x and root $\hat{u}$. Each layer of green boxes corresponds to a secondary structure, with each box being a memory block. The dots are the versions, with red dots denoting spanning versions. The gray arrows are the pointers to the spanning predecessor, with all but three arrows omitted for simplicity. The blue arrows are the links described in property **P6**. The top layer is the secondary structure for the node r, and the middle layer is for the node r' on the path x defines through the tree. The yellow arrows trace the returned value through the recursion. Version c is returned at the base of the recursion, and at the middle layer, c is further returned as its spanning predecessor is equal to the candidate b from this layer. At the top layer, a is then returned, as the spanning predecessor of c is not a, resulting in a being the correct predecessor of the node v.

5.1.3 Correctness

Let $s = (v_s, x_s, y_s)$ be the segment that should be reported by the ray shooting query and let $\{D_1, D_2, \ldots, D_d\}$ be the secondary structures on the path to the elementary interval containing x starting with D_1. By property **P3**, v_s is stored in one of these as a spanning version, defined for the following as the i'th. By property **P4**, v_s is stored in all secondary structures D_j where $j \leq i$ and thus setting $v = \mathit{pred}(v, D_{u_c})$ in the recursion preserves that $v \geq v_s$. This guarantees that in D_i we find the version we are looking for since here $\mathit{spanning\_pred}(v, D_i) = v_s$. We maintain the best candidate version by performing the check $\mathit{spanning\_pred}(v_u, D_u) = \mathit{spanning\_pred}(v, D_u)$ which is equivalent to checking $v_u \geq \mathit{spanning\_pred}(v, D_u)$. Finally, we use $\mathit{findAbove}(v_{u_c}, D_u)$ to maintain a pointer to the location of the candidate version in the secondary structure at the current level of the recursion, which is needed for the calls to $\mathit{spanning\_pred}(v_u, D_u)$.

5.1.4 Efficiency

The depth of the recursion is given by the height of the segment tree which is $T_h = \mathcal{O}(\log_B S)$. At every level, the number of I/Os performed depends on the efficiency of the functions *spanningPred*, *pred* and *findAbove* when called on some version v. The function *spanningPred* takes $\mathcal{O}(1)$ I/Os since it follows a single pointer. By property **P6** the two versions immediately surrounding v that are bridges to the next (previous) level create a loop of length $\mathcal{O}(1)$ in the blocked linked lists. Since the last two functions *pred* and *findAbove* can be performed by traversing the loop and comparing versions only for equality, then ray shooting requires at most $\mathcal{O}(\log_B S)$ I/Os.

5.1.5 Space Usage

The space usage is dominated by the secondary structures. The total number of versions stored in all secondary data structures without the cascading from bridges (property **P6**) is $\mathcal{O}(SB \log_B S)$, since properties **P2** and **P4** imply that every version is stored in at most $2\delta = B$ secondary structures at every level of T. With cascading, every B'th element of a secondary structure cascades down to each of the $\delta \leq B/2$ children. However, since $\delta/B \leq 1/2$ this is geometrically decreasing and the total number of versions increases by no more than a factor 2. Since the blocks in the linked list contain $\Omega(B)$ versions the total number of blocks required is $\mathcal{O}(S \log_B S)$.

5.2 Dynamic Ray Shooting

We now describe how to maintain the structure under insertions such that it still preserves all the properties described for the static structure. The main challenges are as follows:

1. The secondary structures which are blocked linked lists must handle insertions of versions anywhere since segments are not inserted in order. We handle this by dynamically splitting blocks in the secondary structure when they become too large. This requires updating the surrounding bridges.

2. New segments will be inserted as spanning in a number of secondary structures. To maintain property **P5** the predecessor pointer of surrounding non-spanning versions must be updated.

3. New segments may split up to 2 existing elementary intervals into 4 new. To prevent the segment tree from becoming unbalanced, we implement it using a weight-balanced B-tree, which furthermore allows us to rebuild the secondary structures when nodes split.

4. When a node u is split into u_{left} and u_{right}, non-spanning versions in u may become spanning in either u_{left} or u_{right}. These versions must no longer be stored in the children of u to preserve property **P2**.

5. When a node u is split, the bridges of u must be removed from the children of u. This may cause blocks to be merged and further bridges have to be removed.

Note that any insertion or deletion of a version in a secondary structure may cascade as bridges need to be maintained. We first handle challenges 1 and 2, where insertions do not create new elementary intervals meaning that no splits occur in the segment tree.

5.2.1 Insertions without new Elementary Intervals

Here we consider the insertion of a segment (v, x, y), where the interval can be represented in the existing segment tree T of height $\mathcal{O}(\log_\delta S)$. As mentioned, we assume that we know where the segment should be inserted among the existing segments. That is, right after the predecessor of v among the versions represented in T.

To insert versions anywhere in the secondary structures we slightly change the size of the blocks. Instead of having a fixed size of B, we maintain the invariant that blocks hold $[B, 2B]$ versions, and immediately split them whenever they become too large. Similarly to how array doubling is normally analyzed, we place a potential of $\Theta(1)$ I/Os on each version after the first B. Now when a block b is split into b and b_{new}, we maintain property **P6** by letting the first version in b_{new} become a bridge. The new bridge is inserted in at most δ blocks, each of which may then also have to be split. However, since $\delta = B/2$, we free $\Omega(B)$ potential for every block that is split. Apart from handling the new bridge, we also have to update the incoming bridges to versions in b_{new}, of which there are at most B. Finally, we have to update the pointers described in property **P5** to spanning versions in b_{new}. If the new version inserted in b is non-spanning, then by using one level of indirection such that all non-spanning versions with the same spanning predecessor share a pointer, again at most B pointers must be updated. Thus, if all the versions are non-spanning then insertions require amortized $\mathcal{O}(B \log_B S)$ I/Os.

However, $\mathcal{O}(\delta \log_\delta S)$ versions are inserted in blocks as spanning for every insertion. Even using indirection to move many pointers at once, if a spanning version is inserted in the middle of k non-spanning versions, then up to $p \leq k/2$ pointers must be updated. However, since the pointers that must be updated are in consecutive blocks in the linked list only $\mathcal{O}(p/B)$ I/Os are needed. We now let every sequence of consecutive non-spanning versions of length k have a potential of $(1/B)k \log_2 k$ and consider a sequence of length k that is split by a spanning version into two sequences of length l and r where $l + r \leq k$[1]. Let $i = \min\{l, r\}$ such that $i \leq k/2$. The potential for the i elements then decreases by at least:

$$\frac{i \log_2 i - i \log_2 k}{B} \leq \frac{i \log_2 \frac{k}{2} - i \log_2 k}{B} = -\frac{i}{B} .$$

This is sufficient to update the pointers of the smaller side as well as the indirect pointer of the larger side in amortized $\mathcal{O}(1)$ I/Os. An illustration of the indirect pointers in the predecessor pointer can be seen in Figure 9. When a non-spanning version is inserted the potential increases by:

[1] $l + r = k - 1$ only if a present non-spanning version becomes spanning.

Figure 9: The indirection used to maintain pointers from non-spanning versions to the predecessor among the spanning versions. Here we have three consecutive sequences of non-spanning versions of length 8, 2 and 3 yielding a potential of $(1/B)\,(8\log_2 8 + 2\log_2 2 + 3\log_2 3)$.

$$\frac{(k+1)\log_2(k+1) - k\log_2 k}{B} \leq \frac{2+\log_2 k}{B} = \mathcal{O}\left(\frac{\log_2 S}{B}\right) = \mathcal{O}(\log_B S)\ .$$

Thus, for every split, we further have to place $\mathcal{O}(\delta \log_B S)$ potential, due to new bridges introducing non-spanning versions in the structure. We accommodate for this by increasing the potential on versions in a block from $\Theta(1)$ to $\Theta(\log_B S)$ such that we free $\Omega(B \log_B S)$ potential for every split. The change also increases the amortized cost of insertions to $\mathcal{O}\big(\delta \log_B^2 S\big)$ I/Os, but we have managed to maintain all the properties required for ray shooting queries as long as no new elementary intervals are created.

5.2.2 Weight-Balanced Segment Tree

In this section, we consider the insertion of segments that create new elementary intervals. To maintain the secondary structures together with the depth of the segment tree T we use a weight-balanced B-tree [5] to implement the segment tree. We make the simplifying assumption that the endpoints of all segments are unique. One can simulate this by ordering the endpoints first by their value and secondly by the associated version. That is segment $s = (v, x, y)$ yields endpoints (x, v) and (y, v). In general for two different endpoints $e_1 = (x_1, v)$ and $e_2 = (x_2, w)$ then $e_1 < e_2 \iff (x_1 < x_2) \vee (x_1 = x_2 \wedge v < w)$. Note that the comparison $v < w$ just needs to be some total order on the versions, and it is not the ordering according to the global version list. For a ray shooting query (x, v) the x-value is replaced by (x, v_{max}) where v_{max} is some version larger than all other versions in the total order.

We first parameterize our weight-balanced B-tree T and recall some basic properties of weight-balanced B-trees. For T we choose the branching parameter $a = B/8$ (for $a \geq 4$) and leaf parameter $k = B$, which ensures that all nodes of T have degree at most $4a = B/2$ ([5] Lemma 3.4). This ensures that excess space usage from bridges is still geometrically decreasing. The weight $w(u)$ of a node u is defined as the number of elements stored in the leaves of the subtree rooted at u and thus equal to the sum of the weight of the children of u. Any node u of T at level l (starting at level 0 at the leaves) has weight $(1/2)a^l k \leq w(u) < 2a^l k$, excluding the root where the lower bound does not necessarily hold. Finally, Lemma 3.7 of [5] states that whenever a node u is split then $\Omega(w(u))$ inserts must have been made in the subtree rooted at u since the last split of u. Since we allow every insertion to take amortized $\mathcal{O}\big(B \log_B^2 S\big)$ I/Os then we can free $\Omega(w(u)B \log_B S)$ potential when node u is split. This follows since we can increase the potential of the $\mathcal{O}(\log_B S)$ nodes on the path from the root of T to the two endpoints by $\mathcal{O}(B \log_B S)$. Rebuilding the secondary structures around the node u that is split takes time proportional to the size of the secondary structure D_u. The following lemma relates the weight of a node and the size of its secondary structure.

Lemma 1. *Let u be a node in T with associated secondary structure D_u, and let $|D_u|$ denote the number of versions stored in D_u. Then $|D_u| = \mathcal{O}(aw(u))$.*

Proof. Let I_u be the interval defined by D_u. Versions are stored in D_u in the following three cases:

1 If the segment has an endpoint in I_u (property **P4**). This contributes at most $w(u)$ versions.

2 If the segment spans I_u but not the interval of the parent of u (property **P2**). This implies the segment has an endpoint in one of the other children of the parent $u.p$ of u. Thus, this contributes at most $w(u.p) \leq 4aw(u)$ versions.

3 The version is stored in the secondary structure $D_{u.p}$ of the parent of u and has propagated down due to the bridges (property **P6**). This contributes at most $|D_{u.p}|/B$ versions.

Due to the third case, we use induction in the levels of T starting at the root to show that $|D_u| \leq caw(u)$ for some constant $c > 0$. The base case is the root $\hat{u}$ where $|D_{\hat{u}}| \leq w(\hat{u})$. For the induction step for some node u we sum up the contribution from the three cases and apply the induction hypothesis on $|D_{u.p}|$ from case 3:

$$|D_u| \leq w(u) + 4aw(u) + \frac{|D_{u.p}|}{B} \tag{5}$$
$$\leq w(u) + 4aw(u) + \frac{caw(u.p)}{B} \tag{6}$$
$$\leq w(u) + 4aw(u) + \frac{4ca^2w(u)}{B} \tag{7}$$
$$= w(u)\left(1 + 4a + \frac{1}{2}ca\right) \tag{8}$$
$$\leq caw(u) \tag{9}$$

The third inequality holds since $w(u.p) \leq 4aw(u)$. The final inequality holds for $c \geq (2/a) + 8 \geq 8.5$. □

Now that we have established a relationship between the size of the secondary structure of a node and its weight, we describe how to rebuild the secondary structures (challenges 3,4,5). Let u be a node that is split into u_{left} and u_{right}, and let u_c be either one of them. The secondary structure of D_{u_c} should include the following:

1. The bridges from $u.p$.
2. The versions in its children. If the version is spanning in all the children it should be spanning in D_{u_c}
3. The spanning versions of u should also be spanning versions in D_{u_c}.

We identify the versions described above by scanning D_u and D_z for one child z of u_c at a time[2] and increment a counter for every version in D_u also appearing in D_z, except for the bridges from u. If u_c has d children then every version with a count of d must appear as spanning in all its children. It should then be included in D_{u_c} as a spanning version. Similarly, any version with a count greater than 0 but less than d will be a non-spanning version in D_{u_c}. All bridges from $D_{u.p}$ are simply copied to D_{u_c}. Using Lemma 1 we find the number of I/Os required for a single child z to be:

$$\mathcal{O}\left(\frac{|D_z| + |D_u|}{B}\right) = \mathcal{O}\left(\frac{aw(z) + aw(u)}{B}\right) = \mathcal{O}\left(\frac{w(u) + aw(u)}{4B}\right) = \mathcal{O}(w(u))$$

Since u has at most $4a$ children the total number of I/Os required is $\mathcal{O}(w(u)B)$. The potential of the new secondary structure starts at $\Theta(w(u)B\log_B S)$ due to each version having a potential of $\Theta(\log_B S)$.

In the children of u_c we must 1) remove the versions that are spanning in all the children 2) insert the new bridges from u_c 3) remove the old bridges from u. The total number of insertions and deletions is bounded by $\sum_{z=child(u_c)} |D_z| = \mathcal{O}(w(u)B)$ and since we free $\Omega(w(u)B\log_B S)$ potential we have $\Omega(\log_B S)$ I/Os for each of them. Based on the discussion at the end of Section 5.2.1 this is exactly what we need. The constant in the potential on subtrees for weight balance is chosen sufficiently bigger than the constant for potential on versions in blocks. To handle that versions are now also deleted we extend versions in blocks to also have potential $\Theta(\log_B S)$ when the block has few elements. Again, this is similar to normal array doubling.

5.3 Combining with geometric structure

We now briefly describe how our point location structure combines with the geometric construction from Sections 2 and 3 based on the following corollary:

Corollary 2. *Let $\overline{N}$ be the parameter from Theorem 1 giving a bound on the number of updates to the fully persistent search tree and let S from Theorem 3 be the number of segments inserted in the point location structure due to these updates. Then the point location structure performs queries in worst-case $\mathcal{O}(\log_B \overline{N})$ I/Os, insertions in amortized $\mathcal{O}(B\log_B^2 \overline{N})$ I/Os and uses space $\mathcal{O}(\overline{N}/B)$ blocks.*

Proof. In Section 3.8 we show that the number of primitive rectangle transformations is $\mathcal{O}(\overline{N}/R)$ and each of them triggers at most one insertion in the point location structure. Thus, $S = \mathcal{O}(\overline{N}/R)$ which in Theorem 4 results in the claimed bounds. □

Therefore, when using the point location structure for our rectangular partition the bounds match the requirements in Section 2.4.

[2]We scan through only one child at a time to avoid using the tall cache assumption.

6 Partitioning the Version Tree

In this section, we prove the following theorem which describes how to improve the I/O bounds in Theorem 2, such that they depend on the size N_v of the accessed version v instead of the upper bound $\overline{N}$ on the total number of updates.

Theorem 5. *Assume we have a data structure for external-memory fully-persistent search trees supporting a sequence of at most $\overline{N}$ updates, for a constant $\overline{N}$, that supports* Insert, Delete *and* Clone *in amortized $\mathcal{O}(\log_B \overline{N})$ I/Os,* Search *in worst-case $\mathcal{O}(\log_B \overline{N})$ I/Os, and* Range *in worst-case $\mathcal{O}(\log_B \overline{N} + K/B)$ I/Os, and uses space linear in the number of updates. Then there exist external-memory fully-persistent search trees supporting* Insert, Delete *and* Clone *in amortized $\mathcal{O}(\log_B N_v)$ I/Os,* Search *in worst-case $\mathcal{O}(\log_B N_v)$ I/Os, and* Range *in worst-case $\mathcal{O}(\log_B N_v + K/B)$ I/Os, and uses space linear in the number of updates.*

The basic idea of our approach is to split the version tree into smaller version trees by cutting out subtrees after a certain number of updates, such that all versions in a version tree have approximately the same size. Note this introduces an extra level of indirection from versions to the version tree they are represented in. This level of indirection also needs to be maintained but is not the bottleneck.

To be more precise, let T be a version tree rooted at version v_0. We let $N_0(T)$ be the initial number of insertions in version v_0 before other updates are performed at version v_0. We let $upd_T(v_0)$ denote the number of insertions, deletions and clones (i.e., $|T| - 1$) in T, excluding the initial $N_0(T)$ insertions in version v_0. For a subtree T_v rooted at a non-root node v, we let $upd_T(v)$ denote the number of insertions, deletions and clones (i.e., $|T_v|$) in T_v.

Let c be a constant, where $0 < c < 1$. We maintain the invariant $upd_T(v_0) \leq cN_0(T)$. Crucially, this ensures that $|N_v - N_0(T)| \leq cN_0(T)$ for any version v in T. We split T when $upd_T(v_0) = \lfloor cN_0(T) \rfloor$. We say that a version v is *small* if $N_v < \frac{2(2-c)}{c(1-c)}$. Small versions can be maintained naively as a list of values. If a new version tree arising from a split has a small version as the root, we get rid of the tree and instead represent all small versions in the tree naively. Each version in the tree that is not small becomes a version tree by itself. As soon as a version that is maintained naively is not small, it is converted into a version tree.

Choosing $\overline{N} = (1 + c)N_0(T)$ ensures $\overline{N} \leq \frac{1+c}{1-c} N_v$ for all versions v in T, i.e., $\mathcal{O}(\log_B \overline{N}) = \mathcal{O}(\log_B N_v)$ and the bounds in Theorem 5 follow, ignoring the cost for splitting version trees. The split of T will result in four version trees T', T_H, T_L and T_R, some of which may be empty, and we rebuild each of them by simply performing all the updates again to an initial empty version tree. We assume that the version tree and the history of all insertions and deletions for each version are maintained explicitly so that we can easily repeat all operations. This introduces a constant space overhead for each update.

To split T, we first find a *heavy subtree* T_v rooted at a node v with many updates, but where none of the subtrees at the children are heavy. This ensures that by cutting T_v from T many updates must be performed before the next split of T. See Figure 10. More formally, the node v must satisfy the following,

1. Node v is *heavy*, that is $upd_T(v) > \frac{c}{2} N_0(T)$, and
2. $upd_T(w) \leq \frac{c}{2} N_0(T)$ for all children w of v.

We can always find a node v satisfying the conditions above by following a path of heavy nodes from the root v_0 towards the leaves since the leaves always satisfy the second condition. We now describe the four resulting trees and show that a linear number of updates must be performed in each of them before they are split again.

We construct the subtree $T' = T \setminus T_v$ only if $v_0 \neq v$, where T_v is the subtree of T rooted at v. In version v_0 of T, there might be values that have been inserted but have been canceled again by deletions. In T' the initial insertions in version v_0 are all the insertions in version v_0 of T, including the initial insertions, that have not been canceled by a deletion. Version v_0 of T' contains no deletions. For all other versions in T', the updates are the same as in T. The number of insertions left in v_0 in T' defines the new $N_0(T') \geq N_0(T) - (upd_T(v_0) - upd_T(v)) > N_0(T) - (cN_0(T) - \frac{c}{2} N_0(T)) = (1 - \frac{c}{2}) N_0(T)$, as all updates, but the updates in v may be deletions in v_0. Similarly, the number of updates in T' is now $upd_{T'}(v_0) = upd_T(v_0) - upd_T(v) < cN_0(T) - \frac{c}{2} N_0(T) = \frac{c}{2} N_0(T)$. Therefore

$$\frac{upd_{T'}(v_0)}{N_0(T')} < \frac{\frac{c}{2} N_0(T)}{(1 - \frac{c}{2}) N_0(T)} = \frac{c}{2 - c} ,$$

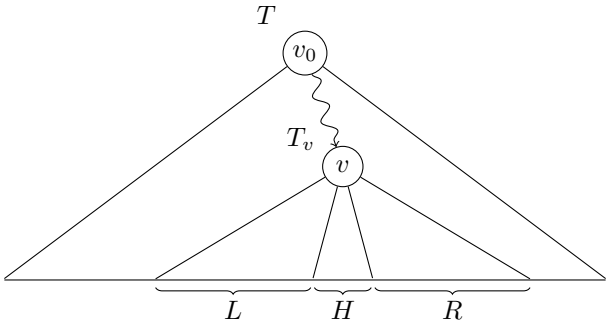


Figure 10: The version tree T rooted at node v_0 that is split by cutting out the subtree T_v rooted at the node v.

and a gap of at least $\lfloor cN_0(T')\rfloor - upd_{T'}(v_0) \geq \left(c - \frac{c}{2-c}\right) N_0(T') - 1$ future updates must occur in T' before we need to split T'. Note $c - \frac{c}{2-c} > 0$ for all $0 < c < 1$. If v_0 is not small, i.e., $\frac{2(2-c)}{c(1-c)} \leq N_0(T')$, we always have a gap of at least one update. For $c = \frac{1}{2}$ the gap is of size $\frac{1}{6}N_0(T') - 1$ and $12 \leq N_0(T')$.

We split the subtree T_v into at most three version trees T_H, T_L, and T_R, depending on the number of children of v. Version v exists in all three version trees, but when an operation subsequently refers to version v, we let it refer to version v in T_H. The version tree T_H consists of v and the subtree at the child of v containing the most updates. If v has no children, T_H is still created but only contains v. Next, we greedily partition the remaining children of v into two sets L and R, such that the total number of updates in each is at most $\frac{c}{2}N_0(T)$. We then create the two subtrees T_L and T_R, which are rooted at v with children L and R, respectively. These version trees are only created if they have at least one child. Similarly to the case for the root v_0 of T', for the root v of T_H, T_L, and T_R, we examine all the insertions and deletions on the path from v_0 to v in T and keep only the insertions that are not canceled by a deletion on the path. The resulting set of insertions is the initial set of insertions for v of size $N_0(T_H) = N_0(T_L) = N_0(T_R)$. We define d to be the number of deletions on the path from v to v_0. Since $upd_{T_H}(v) \leq \frac{c}{2}N_0(T)$ by the second condition in the definition of v, and $upd_{T_L}(v) = \sum_{w\in L} upd_T(w) \leq \frac{c}{2}N_0(T)$ (similarly for T_R) by the choice of L and R, the analysis becomes the same for all three version trees with root v. Here we consider the analysis for T_L and note that $d \leq upd_T(v_0) - \sum_{w\in L} upd_T(w)$. The number of initial insertions in v in T_L is exactly the number of initial insertions in v_0 in T, that are not deleted by the d deletions on the path from v to v_0, i.e., $N_0(T_L) = N_0(T) - d$. Therefore

$$\begin{aligned}\frac{upd_{T_L}(v)}{N_0(T_L)} &\leq \frac{\sum_{w\in L} upd_T(w)}{N_0(T) - \left(upd_T(v_0) - \sum_{w\in L} upd_T(w)\right)} \\ &\leq \frac{\frac{c}{2}N_0(T)}{(1-c)N_0(T) + \frac{c}{2}N_0(T)} = \frac{c}{2-c}\,.\end{aligned}$$

Thus, we get the same gap as for T'.

Finally, we argue about the resulting I/O and space bounds. Each of the four resulting version trees created by a split contains at most $\left(1 + \frac{c}{2}\right) N_0(T)$ updates, i.e., by assumption can be constructed using $\mathcal{O}(N_0(T)\log_B N_0(T))$ I/Os. Since there have been performed at least $\left(c - \frac{c}{2-c}\right) N_0(T) - 1$ updates since T was created, we can charge the cost of splitting T to these updates, yielding an additional amortized $\mathcal{O}(\log_B N_0(T))$ cost per update. Since all versions in T have size at least $\left(1 - \frac{c}{2}\right) N_0(T)$, we can restate the I/O cost as amortized $\mathcal{O}(\log_B N_v)$. Similarly, the additional $\mathcal{O}(N_0(T_H)/B)$ blocks of space overhead when splitting T for introducing v and all $N_0(T_H)$ initial updates to v in all three versions trees T_H, T_L and T_L, can be charged to the at least $\left(c - \frac{c}{2-c}\right) N_0(T) - 1$ updates to T since T was constructed, i.e., the space usage remains linear.

In the above analysis we showed that by setting $c = \frac{1}{2}$, we are guaranteed for each of the resulting trees, e.g., T', at least $\frac{1}{6}N_0(T') - 1$ updates to T' are required before T' is required to be split. Any choice of $0 < c < 1$ works, and at least $\left(c - \frac{c}{2-c}\right) N_0(T') - 1$ updates are required before a split. The best choice is $c = 2 - \sqrt{2}$, where the lower bound becomes $(3 - 2\sqrt{2})N_0(T') - 1 = 0.1715N_0(T') - 1$, that is slightly better than the $\frac{1}{6}N_0(T') - 1 = 0.1667N_0(T') - 1$ lower bound achieved by setting $c = \frac{1}{2}$.

7 Lazy Clones

In this section we prove the following theorem which describes how to improve the amortized I/O bound on CLONE operations in Theorem 5, such that CLONE operations use worst-case constant I/Os.

Theorem 6. *Assume we have a data structure for external-memory fully-persistent search trees that supports* INSERT, DELETE *and* CLONE *in amortized* $\mathcal{O}(\log_B N_v)$ *I/Os,* SEARCH *in worst-case* $\mathcal{O}(\log_B N_v)$ *I/Os, and* RANGE *in worst-case* $\mathcal{O}(\log_B N_v + K/B)$ *I/Os, and uses space linear in the number of updates. Then there exist external-memory fully-persistent search trees supporting* INSERT *and* DELETE *in amortized* $\mathcal{O}(\log_B N_v)$ *I/Os,* CLONE *in worst-case* $\mathcal{O}(1)$ *I/Os,* SEARCH *in worst-case* $\mathcal{O}(\log_B N_v)$ *I/Os, and* RANGE *in worst-case* $\mathcal{O}(\log_B N_v + K/B)$ *I/Os. The space usage is linear in the number of updates.*

The idea of the construction to reduce the I/O bound for CLONE operations, is to postpone the actual cloning to the first update to the version and to charge the cost for the cloning to the later update instead. Any version v, which has not been the subject of any updates, must contain the same values as the (locked) parent version u it has been cloned from. Therefore, any clone or query performed on version v can be performed on version u. Hence version v does not need to be explicitly created in the structure. When version v is cloned, it will become locked, and a new unlocked version w is created that is identical to both v and u. Thus, the same result is obtained as if w was cloned from u.

In this way, *lazy clones* can be implemented using a single layer of references. Each version is either a *real* version existing in the underlying structure or a *lazy* version pointing to a locked real version containing the same values. When making a clone w of a lazy version v pointing to a real version u, version v can no longer receive updates and is locked. Version v remains lazy and is never explicitly constructed. The new version w becomes lazy, pointing to the real version u. As a CLONE operation always creates a lazy version from some real version, and a real version can be found by traversing at most 1 pointer, then CLONE operations use worst-case $\mathcal{O}(1)$ I/Os. Any INSERT or DELETE operation must first check if the operation is performed on a real or lazy version, which uses $\mathcal{O}(1)$ additive I/Os. If the version is lazy, it must first be made real, which uses amortized $\mathcal{O}(\log_B N_v)$ I/Os, by performing a CLONE operation on the underlying data structure given to the construction. After this, the update can be performed in amortized $\mathcal{O}(\log_B N_v)$ I/Os, resulting in total amortized $\mathcal{O}(\log_B N_v)$ I/Os. Similarly, SEARCH and RANGE operations on lazy versions are instead performed on the equivalent real versions. The overhead is $\mathcal{O}(1)$ additive I/Os. Finally, storing the single layer of references from lazy versions to real versions requires only additive linear space. This concludes Theorem 6.

8 Conclusion and Future Work

This paper presents external-memory fully-persistent B-trees with I/O bounds (Theorem 1) matching those of classical B-trees [6]. A natural open question is whether this result can be extended to the update-query trade-off regime by buffering updates as was done for classical B-trees. Adopting our solution seems plausible but nontrivial since it likely requires buffering the point location structure. In the B^ε tree, queries can be performed efficiently since all the relevant buffered updates are on the path in the tree from the root to the query position. However, that would not be the case in the segment tree, which our point location structure is built upon. Another direction is to improve the amortized bounds to instead hold with high probability or worst-case. The main obstacle here seems to be how to handle rectangle rebalancing. These open questions are similar to the improvements to the classical B-tree mentioned in Section 1.2, and likely some of those techniques can also be deployed here.

References

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962. English translation by Myron J. Ricci in Soviet Mathematics - Doklady, 3:1259–1263, 1962.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

[3] Lars Arge, Gerth Stølting Brodal, and S. Srinivasa Rao. External memory planar point location with logarithmic updates. *Algorithmica*, 63(1):457–475, 2012. doi:10.1007/s00453-011-9541-2.

[4] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8, 2003. doi:10.1145/996546.996549.

[5] Lars Arge and Jeffrey Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32:1488–1508, 09 2003. doi:10.1137/S009753970240481X.

[6] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683.

[7] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996. doi:10.1007/s007780050028.

[8] Michael A. Bender, Rathish Das, Martin Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing without cascades. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 650–669. SIAM, 2020. doi:10.1137/1.9781611975994.40.

[9] Michael A. Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. Write-optimized skip lists. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, page 69–78, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3034786.3056117.

[10] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. SODA '03, page 546–554, USA, 2003. Society for Industrial and Applied Mathematics.

[11] Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. External memory fully persistent search trees. In *Proceedings of the 53rd Annual ACM Symposium on Theory of Computing*, STOC '23, pages –, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3564246.3585140.

[12] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Fully persistent B-trees. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 602–614. SIAM, 2012. doi:10.1137/1.9781611973099.51.

[13] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Fully persistent B-trees. *Theoretical Computer Science*, 841:10–26, 2020. doi:10.1016/j.tcs.2020.06.027.

[14] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986. doi:10.1137/0215051.

[15] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986. doi:10.1007/BF01840440.

[16] Rathish Das, John Iacono, and Yakov Nekrich. External-memory dictionaries with worst-case update cost, 2022. arXiv:2211.06044.

[17] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Transactions on Algorithms*, 3(2), May 2007. doi:10.1145/1240233.1240236.

[18] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM. doi:10.1145/28395.28434.

[19] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

[20] Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms*, 5(3):28:1–28:51, July 2009. doi:10.1145/1541885.1541889.

[21] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.

[22] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. doi:10.1007/BF00288968.

[23] Sitaram Lanka and Eric Mays. Fully persistent B+-trees. *SIGMOD Records*, 20(2):426–435, April 1991. doi:10.1145/119995.115861.

[24] David B. Lomet and Betty Salzberg. Exploiting a history database for backup. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 380–390, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. URL: https://vldb.org/conf/1993/P380.PDF.

[25] J. Ian Munro and Yakov Nekrich. Dynamic planar point location in external memory. In Gill Barequet and Yusu Wang, editors, *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA*, volume 129 of *LIPIcs*, pages 52:1–52:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.SoCG.2019.52.

[26] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986. doi:10.1145/6138.6151.

[27] Peter J. Varman and Rakesh M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997. doi:10.1109/69.599929.

[28] Dan E. Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing*, 14(1):232–253, 1985. doi:10.1137/0214019.

Chapter 8

Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location

Space-Efficient Functional Offline-Partially-Persistent Trees with Applications to Planar Point Location*

Gerth Stølting Brodal, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, and Rolf Svenning

Department of Computer Science, Aarhus University, Denmark
{gerth,rysgaard,jkrs,rolfsvenning}@cs.au.dk

Abstract. In 1989 Driscoll, Sarnak, Sleator, and Tarjan presented general space-efficient transformations for making ephemeral data structures persistent. The main contribution of this paper is to adapt this transformation to the functional model. We present a general transformation of an ephemeral, linked data structure into an offline, partially persistent, purely functional data structure with additive $\mathcal{O}(n \log n)$ construction time and $\mathcal{O}(n)$ space overhead; with n denoting the number of ephemeral updates. An application of our transformation allows the elegant slab-based algorithm for planar point location by Sarnak and Tarjan 1986 to be implemented space efficiently in the functional model using linear space.

Keywords: Data structures · Functional · Persistence · Point location

1 Introduction

The functional model has many well-known advantages such as modulation, shared resources, no side effects, and easier formal verification [4,14]. These advantages are given by restricting the model to only use functions and immutable data. As all data are immutable, side effects in functions are not possible, allowing modules to work independently of the context they are used in and reducing the complexity of formal verification [14].

In 1999 Okasaki [19] gave a seminal work on techniques for designing efficient (purely) functional data structures, and our result follows this line of research. Adapting existing data structures to the functional model is non-trivial since modifications are prohibited. However, it also means that updates do not destroy earlier versions of the data structure making functional data structures inherently persistent, but not necessarily space efficient. The focus of this paper is to adapt existing imperative techniques for persistence to the functional model in a space-efficient manner.

We introduce a purely functional framework that adapts classical tree structures to support offline partial persistence with an additive overhead. By offline

* Work supported by Independent Research Fund Denmark, grant 9131-00113B.

Table 1: Previous and new results for planar point location, where † are expected bounds and * are results based on persistent data structures.

Reference	Construction	Query	Space	Model
David Kirkpatrick [17]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative
Seidel [26]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative†
Dobkin and Munro [11]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n \log n)$	Imperative*
Richard Cole [8]	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative*
Sarnak and Tarjan [24]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Imperative*
Sarnak and Tarjan [24]	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$	Functional*
New	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Functional*

partial persistence, we mean that all updates are made before queries. This restriction allows us to store update information without immediately being able to handle queries efficiently. A data structure that is not persistent, i.e., does not support queries and updates in previous versions, is said to be *ephemeral*. We show that by recording when an ephemeral tree structure is updated, we can build a query structure that can efficiently answer queries to previous versions of the structure. In the imperative paradigm, it is possible to efficiently interweave updates and queries [12,24], but in the functional paradigm, it incurs a multiplicative logarithmic space overhead, which we show how to circumvent, in the offline setting.

Planar point location is a classic computational geometry problem [13,17,24]. Given a planar straight-line graph with n edges (interchangeably line segments) and report the region containing a query point q. The task is to create a data structure that supports these queries while minimizing the construction time, query time, and space of the data structure. Sarnak and Tarjan [24] showed that the planar point location problem can be solved elegantly using *partially-persistent* sorted sets, that in the functional setting can be solved with balanced search trees using path copying resulting in a space usage of $\mathcal{O}(n \log n)$. As an application of our technique, we show how the algorithm of Sarnak and Tarjan can be implemented in the functional model to only use space $\mathcal{O}(n)$. For an overview of results for the planer point location problem see Table 1.

Below we state sufficient conditions for a functional or imperative data structure to be augmented with a functional support structure that supports offline partial persistence queries.

Definition 1 (`TUNA` conditions).

- `T:` *The data structure forms a rooted tree of constant degree d.*
- `U:` *Updates create $\mathcal{O}(1)$ new edges and nodes.*
- `N:` *No cycles are created by updates when considering the edges that have been created across all versions.*
- `A:` *Attribute values of nodes are static, i.e., the information stored in a node is not changed after its insertion. This does not include fields pointing to the children.*

This definition is not too restrictive, we show that binary search trees (BST), Treaps [3], Red/Black Trees [5], and Functional Random Access Arrays [18] all can be modified to satisfy Definition 1, without asymptotically significant overhead. In Section 5 we discuss how some of these requirements can be relaxed or generalized further.

Theorem 1. *For any ephemeral, linked data structure that satisfies the* `TUNA` *conditions, an equivalent, functional, offline-partially-persistent data structure preserving the asymptotic update and query times, can be created, with an additive construction overhead of time* $\mathcal{O}(n \log n)$ *and space* $\mathcal{O}(n)$ *for a series of* n *updates.*

The main idea behind Theorem 1 is to store the update information in a list, including edge insertion and deletion timestamps. After all the updates have been applied, a bottom-up topological sort produces a directed acyclic graph (DAG) that supports queries to any previous version of the data structure. This is essentially implementing the node copying approach of [24], while carefully avoiding the creation of cycles.

Building upon [24], Theorem 1 immediately implies a state-of-the-art functional planar point location solution, summarised in the following corollary.

Corollary 1. *There exists a purely functional solution to the planar point location problem with construction time* $\mathcal{O}(n \log n)$*, query time* $\mathcal{O}(\log n)$*, and space* $\mathcal{O}(n)$*.*

We implemented unbalanced binary search trees in the purely functional programming language Haskell and report on some experiments in Section 6.

1.1 Persistence

A data structure is said to be *persistent* if it is possible to query previous versions of it and *ephemeral* if it is only the current version that is available. A *partially-persistent* data structure is *persistent* and allows updates only to the latest version. The stronger notion of *full persistence* implies that any version can be both queried and updated. An update to a persistent data structure never changes an existing version, but instead creates a new version derived from the version the update is applied to. In this way, the different versions of the partial persistent structure form a version list, whereas full persistence forms a version tree. General transformations to make data structures persistent were studied by Driscoll et al. [12] and Overmars [21,22]. In this paper, we focus on *offline* partial persistence for linked data structures, where all updates are performed before all queries, which was also explored in [11].

A naive idea to achieve partial persistence is to store a copy of every previous version. If the underlying structure is a list, then this approach generates an overhead of $\Omega(n^2)$ space for n insertions into an initially empty list. To improve upon this, the crucial observation is that when structures have a large overlap between updates, it is possible to reuse large parts of the previous versions and

greatly reduce space usage. To achieve linear space, the notions of *fat nodes* and *timestamps* were introduced by Driscoll et al. [12], where each pointer field in a node is replaced by a list of pointers and each pointer in the list has an associated timestamp, denoting when the pointer was updated. Specifically, for a binary search tree, each node will, instead of containing a pointer to the `left` and `right` subtree, contain a list of timestamped `left` pointers and a list of timestamped `right` pointers. An update to a given pointer now adds a new pointer to the pointer list with the current timestamp, resulting in $\mathcal{O}(1)$ space overhead per update. As the pointer in the version is the last in the list, there is no overhead in finding the active pointer in the current version. To locate the correct pointer at a given older timestamp, a binary search on the list of pointers can be performed, which then imposes a multiplicative $\mathcal{O}(\log n)$ overhead on queries.

In [24] the notion of *path copying* was introduced for BSTs, where all nodes on the path to the node being updated are copied. Any existing pointer along the path can then point freely to parts of the old structure. For BSTs, this has a space overhead of the length of the path, i.e., for balanced BSTs an $\mathcal{O}(\log n)$ space overhead per update. It does however impose no overhead on the query time, apart from initially finding the correct root to query.

Thus, the fat node technique has a query time overhead, whereas path copying has a space overhead. By combining these two techniques, it is possible to have no overhead on query time and space. In [12,24] the authors achieve this by introducing the *node copying* technique for partially-persistent general pointer-based data structures. Here nodes are allowed to hold a constant number of additional time-stamped pointers. When some operation requires a node to update a pointer, the timestamp of the old pointer is updated to end at the current time. We denote a pointer that has ended as *expired*. If the pointer is replaced by a new pointer, then the new pointer is placed in one of the free extra pointer slots of the node, with a timestamp starting at the current time. If there is no free pointer slot *node copying* is performed, where the non-expired pointers and the new pointer are copied to a new node. Any pointer in another node to the copied node at the current time must be split, which may cause node copying to cascade up the structure. However, an amortization argument [24] shows that this technique only has an additive $\mathcal{O}(n)$ overhead in space, when the indegree of every node in the underlying structure is constant. Finally, as the number of pointers in each node is constant the overhead on the query time is also constant.

1.2 Persistence and the functional model

The functional programming paradigm is well suited for persistence as stated by Okasaki: "*A distinctive property of functional data structures is that they are always persistent*" [19]. In purely functional programming, there are no side effects and variables are immutable, meaning that any modifications to a structure S are obtained by creating a new structure S' without altering S. In this new structure S', substructures may be references to (immutable) old substructures from S.

Purely functional data structures are consequently particularly interesting when persistence is critical, and it is natural that they are less efficient than imperative data structures, since they inherently solve a more general problem. We note that there has been some work in describing to what degree mutability increases the capabilities (efficiency) of a language, and Pippenger [23] gave an example of a problem with a logarithmic-factor separation under certain conditions. However, for many data structure problems, purely functional solutions have been developed that match their imperative counterparts with only constant overhead. Examples include optimal *confluently* persistent deques which were developed over a number of papers [7,9,16] and optimal priority queues [6].

1.3 Functional vs. space efficient imperative persistence

The fat node and node copying persistence techniques mentioned in Section 1.1 rely upon the imperative paradigm's ability to modify pointers in the nodes in a graph structure where nodes can have multiple ingoing edges. As the functional model cannot mutate pointers, directly translating these solutions leads to significant overhead in the update time, as, even with path copying, all ancestors of an updated node must be remade to point to the newly created node(s). For this reason, we focus our attention on data structures with a tree structure where ancestors appear on a single path to the root.

1.4 Planar point location

Dobkin and Lipton [10] solved the planar point location problem by drawing vertical lines through every node resulting in vertical slabs. For every slab, the line segments spanning the slab are stored in a BST. This method allows efficient queries by performing a binary search horizontally for the slab, and then a binary search vertically in the slab for the region. This gives an overall query time of $\mathcal{O}(\log n)$. The drawback of this method is that each line could potentially be stored in almost every slab, resulting in $\Theta(n^2)$ space. A number of different results [8,13,17,24], show that the space can be reduced to $\mathcal{O}(n)$ (non-functional) without affecting the query time. The solution by Cole [8] is particularly interesting as it exploits that neighboring slabs are very similar, meaning that the problem can be reduced to creating persistent, sorted sets. This observation is vital to the work on persistent search trees by Sarnak and Tarjan [24]. On the other hand, the approach by Kirkpatrick [17] is completely different and is based on repeatedly triangulating the graph and removing a constant fraction of the nodes with degree at most 11. Then a DAG is created bottom-up based on the overlap between two consecutive triangulations. Since the DAG is created bottom-up, similarly to our approach described in Section 3.2, we conjecture that this approach could be adapted to the purely functional model and leave it as an open problem.

2 Initial analysis of binary search trees

In this section, we introduce ephemeral (i.e., non-persistent) unbalanced binary search trees (BSTs) and illustrate how the fat node technique can be adapted to the functional paradigm. The main obstacle in making the adaptation is to avoid cycles between nodes. We solve this by making new copies of nodes that should be moved above their parent. In Section 5 these techniques are extended to cover balanced search trees.

2.1 Ephemeral binary search trees

Any BST node is either an empty leaf or a node containing an element and two pointers to a left and right sub-tree, which are in turn also BSTs. Forthwith, these pointers will be denoted as *edges*. Likewise, let T and T' denote BSTs over a totally ordered set of elements X and let $x \in X$ denote an element. BSTs are ordered such that all elements in the left subtree are smaller than the element in the node, and all elements in the right subtree are larger. BSTs support many operations; we focus on the following three basic operations:

- `Insert`(T, x): Insert x into T and return the resulting tree T'.
- `Delete`(T, x): Delete x from T and return the resulting tree T'.
- `Search`(T, x): Return the smallest element x' in T, such that $x \leq x'$.

All operations can be implemented in time linear in the height of the tree. We call operations that modify the data structure *updates* (`Insert` and `Delete`). We call operations that query the data structure without changing it *queries* (`Search`). When constructing the data structure, we consider a sequence of n updates $(u_1, \ldots, u_n)$ and for version $0 \leq t \leq n$ updates $u_1, \ldots, u_t$ have been applied.

2.2 Fat node binary search trees

In this subsection, we describe how to adapt imperative unbalanced BSTs to adhere to the `TUNA` conditions (Defnition 1), using fat nodes. Most importantly, the `Delete` update is changed slightly from the classic behavior since it can cause nodes to be reordered in the tree.

When performing `Insert`, assuming that x is not present in T, a path to the correct leaf position is found, and a new node, containing x is created at the position of that leaf. The difference between the old and the new tree is a single edge at the bottom of the tree. By adding *creation timestamps* on the edges to represent creation time, queries can detect if a particular edge should be considered to exist in a specific version or not.

When performing `Delete`, if the deleted element x is at the bottom of the tree, then, in effect, the edge e from its parent p to the deleted node ceases to exist. We record this by adding an *ending timestamp* to the edge. If the deleted element x is in some internal node, see Fig. 1 (Left), then a predecessor or

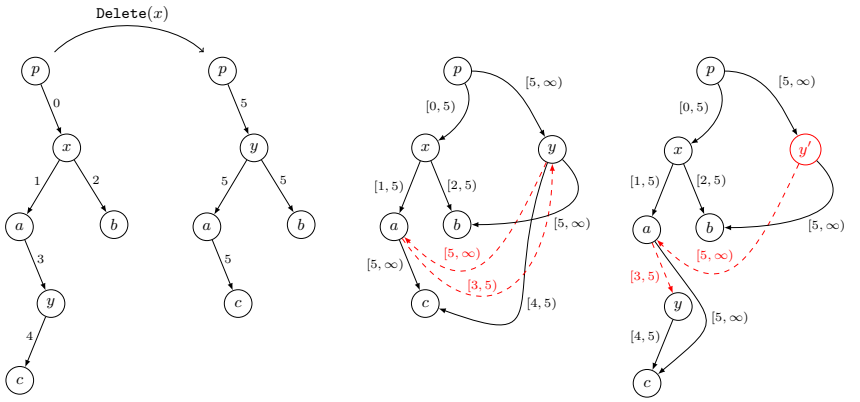


Fig. 1: We let a single number above an edge denote its creation time. In the graphs, the interval above an edge is the living span of the edge. Recall that a persistent search tree query is a timestamp/version and a value. Left: a deletion in a BST. Middle: the resulting graph contains the dashed cycle between y and a. Right: by creating a new copy y' of y the resulting graph remains a DAG and is TUNA compliant.

successor y (depending on implementation) in the subtree rooted at x is found and moved to the deleted spot. This introduces up to four new edges.

When transforming the BST into the functional model, the predecessor cannot be directly moved and reused, as it might be moved above its parent, which would create a cycle in the structure when also considering expired edges, as seen in Fig. 1 (Middle). We avoid creating cycles by replacing x with a copy of its predecessor which breaks the cycle as seen in Fig. 1 (Right). However, it remains a crucial issue that b (and c) still have two parents, when also considering the expired edges, which cannot be updated efficiently in the functional model (see Section 1.3).

3 Freezer and query structure construction

In this section we present our main contribution: the concept of a *Freezer*, which stores update information as it comes in, and how to use it to build a query DAG after all updates have occurred, using fat nodes and node copying. We give an amortized analysis argument that the total number of node copies is linear in the number of updates and from the proof we deduce that the TUNA conditions 1 are sufficient to prove Theorem 1.

3.1 The Freezer

The underlying ephemeral structure forms a tree, but when performing updates using the TUNA-compliant fat node method (see Fig. 3) to get persistence, then

the data structure forms a DAG. This is problematic since in the functional model a DAG cannot be maintained efficiently using path copying, as any node in the data structure which can reach an updated node would also have to be copied to reflect the change. Thus, we store all edges not present in the most recent version of the data structure separately in a list which we call the *freezer*. That is, only the most recent version of the data structure is explicitly maintained. This is sufficient for partial persistence since updates are only allowed in the most recent version. Finally, after all the updates have been applied, the DAG is built bottom-up. For this reason, persistence is restricted to offline.

To store the edges in the freezer, a unique *id* is assigned to each node and each edge in the freezer is stored as a 5-tuple $(id_{\text{from}}, id_{\text{to}}, \mathit{field}, t_{\text{start}}, t_{\text{end}})$. The ids are used to identify the nodes the edge connects, the field denotes which field of the node with id id_{from} the edge originates from (in the BST this is either `left` or `right`), and the time stamps t_{start} and t_{end} denote the *living span* of the edge as the half-open interval $[t_{\text{start}}, t_{\text{end}})$.

The freezer in addition stores which node is the root at any given time, and a map from ids to the value contained in the node with the given id. Further, note that storing the deletion time of an edge can be omitted and instead be read from the starting time of the next edge in the corresponding field of the same node, with the modification that empty fields have an edge to a special `Nil` leaf.

3.2 Offline construction of the fat node query DAG

In this section, we describe how to construct the fat node DAG structure from the expired edges in the freezer and the final non-expired structure.

Any edge that ends in the freezer must have been created in some version of the tree as the result of an update operation. An `Insert` operation creates one new edge, and a `Delete` operation creates at most four new edges. Thus, after all n updates, the freezer contains $\mathcal{O}(n)$ edges. Similarly, each `Insert` and `Delete` creates at most one new node, so the number of nodes is $\mathcal{O}(n)$.

Using Kahn's algorithm [15], which topologically sorts the nodes by repeatedly extracting nodes with outdegree zero, we build the DAG bottom-up. The while loop of the imperative algorithm is replaced by a recursive function in the functional algorithm, for each iteration calling with the new values needed. As mentioned, by copying nodes when they were moved around by the updates we avoid introducing cycles (see Figs. 1 and 3), which is required for Kahn's algorithm. By having a map from node ids to the values contained in the nodes, it is possible to explicitly build the DAG over the edges of the freezer. This creates the DAG of fat nodes.

The construction time of, a non-functional implementation of, Kahn's algorithm is linear in the number of nodes and edges. This however relies on being able to effectively fetch the ingoing and outgoing edges of nodes, and reduce the outdegree of nodes in time $\mathcal{O}(1)$. The construction relies on efficient maps from ids to nodes. As random access is not part of the functional model, maps with $\mathcal{O}(1)$ lookup time and update times do not exist. We instead use balanced trees

which introduces an overhead of $\mathcal{O}(\log n)$ for each of the operations, yielding a DAG construction time of $\mathcal{O}(n \log n)$. The space usage remains $\mathcal{O}(n)$.

3.3 Bounding node outdegree of the query DAG

Having arbitrary outdegree of fat nodes affects the query time, as stated in Section 1.1. However, by limiting the number of extra pointers in each node, limited node copying similar to [24] can be implemented to remove the query overhead, while still maintaining linear space. The difference here is that in [24] the copy is performed during the update phase as soon as there are too many pointers in a node. We restrict ourselves to the offline setting and as such do not need to be able to handle queries before all updates have been applied, this means that we can allow nodes to become arbitrarily fat in the update phase.

Let d be the degree of the underlying ephemeral structure. After the update phase, we handle the high degree by recursively splitting the fat nodes into multiple nodes, each with degrees $\mathcal{O}(d)$ by interleaving the node splitting idea with Kahn's algorithm. As discussed in Section 3.2 we visit the nodes of the freezer bottom-up in topological order. Recall that the freezer contains edges, so nodes are inferred. When visiting a node v, that has an outdegree larger than $d + e$, for a parameter $e = \mathcal{O}(d)$ indicating the extra edges we allow every node to hold, we split it into a left node v_l to represent "the past" and a right node v_r to represent "the future", essentially performing node copying. Note that some edges will be active in both of these nodes and thus are duplicated, similar to regular node copying. We perform the split such that the outdegree of the left node is at most $d+e$, and that the left and right nodes in total represent the original node. If the outdegree of v_r is larger than $d + e$ we recursively split it until v has been split into a number of nodes each of outdegree at most $d + e$. By requiring $1 \leq e \leq cd$, for some constant c, we ensure that the query time is proportional to that of the underlying ephemeral structure since the new nodes will contain $\Theta(d)$ edges. We call this procedure *node copying* and the following lemma shows that the space remains linear in the number of updates n.

Lemma 1. *The number of nodes introduced by* node copying *is* $\mathcal{O}(n)$ *when* $1 \leq e \leq cd$ *for some constant* c.

Proof. We define the potential function

$$\Phi = \sum_v \max\{0, O_v - (d + e)\} ,$$

where O_v is the number of outgoing edges from the node v, and the terms in the sum indicate the number of outgoing edges above the threshold $d + e$. Observe that the potential is nonnegative and that $\Phi = \mathcal{O}(n)$ when initializing *node copying* due to property `U`.

We now analyze how the potential changes when we split a node v, which must have $O_v > d + e$, into a left node v_l and a right node v_r. We consider the edges (consisting of a start and end timestamp) in nondecreasing order by

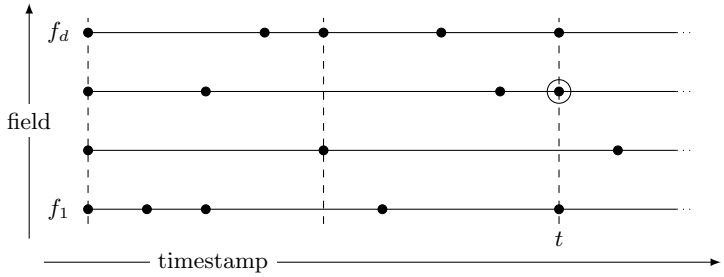


Fig. 2: Illustration of the fields of a node over time. The number of fields is $d = 4$ and $e = 4$. Each horizontal line represents one of the fields, with the dots denoting times when the value of the field change. In other words, the segments between dots represent time intervals where the field is unaltered. The vertical dashed lines are the times when the node is split. The dot with the circle around it is the start of the $(d + e + 1)$th edge (when counting only edges after the previous split) introducing the split at time t. This results in $s_1 = 1$, $s_2 = 2$ and $a = 1$.

their start timestamp. The split is performed at the $(d+e+1)$th smallest start timestamp t. We include all edges with start timestamps strictly less than t in v_l. In v_r we include all edges with an end timestamp strictly larger than t.

Expanding on this, at time t we let s_1 denote the number of edges with start timestamp t among the first $d+e$ edges, and let s_2 be the number of edges with start timestamp t not among the first $d+e$ edges. Together s_1+s_2 is the number of edges changing at time t, or equivalently the number of fields updated at time t. Furthermore, we call an edge *active* if its living span contains the splitting time, that is $t_{start} < t < t_{end}$, and let a denote the number of active edges at time t. See Fig. 2 for an example. Since all nodes have degree d exactly[1], we have $d = s_1+s_2+a$, where $s_2 \geq 1$ from the edge where we perform the split. The number of outgoing edges in v_l is $O_{v_l} = d+e-s_1 \leq d+e$. Likewise, the number of outgoing edges in v_r is $O_{v_r} = O_v - O_{v_l} + a = O_v - d - e + s_1 + d - s_1 - s_2 \leq O_v - (e+1)$, where the inequality follows from $s_2 \geq 1$.

Finally, at time t there is at most one incoming edge to v from a parent v_p, since the underlying structure forms a tree. This edge must potentially be split in two which increases the outdegree of the parent by one. The difference in potential before and after this split is then at most:

$$\begin{aligned}\Delta\Phi &= \Delta\Phi_{v_p} + \Phi_{v_l} + \Phi_{v_r} - \Phi_v \\ &\leq 1 + 0 + \max\{0, O_v - (e+1) - (d+e)\} - (O_v - (d+e)) \,.\end{aligned}$$

Now there are two cases depending on which term is larger in the maximum. We first look at the case when $0 \leq O_v - (e+1) - (d+e)$. We call this case A

[1] Each node has exactly d fields and each field always holds an edge to either another node or to `Nil`.

and here we get $\Delta\Phi \leq -e$. The other case we call case B and here we get $\Delta\Phi \leq 0$ since $\Phi_v \geq 1$. Furthermore, since $O_v - (e+1) - (d+e) < 0$ implies $O_{v_r} \leq O_v - (e+1) < e + d$ then no further splits of v_r are needed. We can now upper bound the number of splits by the number of times these two cases occur. Since the potential never increases, case A can occur at most $\mathcal{O}(n/e)$ times. Next, since we split nodes bottom-up in the topological order, we never add edges to a node after it has been split. Thus, as case B only happens when we perform the last split of a node, it can occur at most $\mathcal{O}(n)$ times, once for each node in the freezer. Combining the two cases we see that the number of splits is $\mathcal{O}(n)$. □

4 Proof of Theorem 1

Let D be a data structure that satisfies the `TUNA` conditions (Definition 1). We identify each node by a unique id and store timestamps with each edge denoting their living span. The freezer, see Section 3.1, stores edges on the form $(id_{\text{from}}, id_{\text{to}}, \mathit{field}, t_{\text{start}}, t_{\text{end}})$, where field denotes the outgoing field of the edge from the node with id id_{from}, and the edge was live in the version interval $[t_{\text{start}}, t_{\text{end}})$. The freezer further records what id the root of the data structure has for each time step, as well as what static value is associated with each id.

Applying an update to the structure can be done without saving the old structure, as condition `A` ensures that all nodes still present contain the same value. Furthermore, condition `T` ensures that the outgoing number of edges is d, leading to a constant number of updates to the freezer for each node updated, resulting in updates having unaltered asymptotic running time.

After applying all updates, recording the relevant information in the freezer, and obtaining tree T, condition `U` ensures that the freezer contains $\mathcal{O}(n)$ edges. For ease of argument, we then enter all edges from T into the freezer. Note that the number of elements in the freezer remains $\mathcal{O}(n)$.

To build the query DAG, apply a modified version of Kahn's algorithm [15] as described in Section 3.2, to perform a bottom-up topological sort of the graph induced by the edges in the freezer in time $\mathcal{O}(n \log n)$. Kahn's algorithm requires that the graph is acyclic, which condition `N` ensures. First, give each node, defined by id, the value stored in the freezer and the edges with matching id_{from}. Second, we employ the fat node technique [12] by allowing nodes to have $d + e$ edges. A fat node can overflow, if it gets more than $d + e$ edges. When an overflow is encountered the node is split into two nodes with the same value and the same parent but only a subset of the edges. Lemma 1 guarantees that splits only cause limited cascading while keeping nodes of degree $\mathcal{O}(d)$ and thus within a constant factor of their degree in the ephemeral structure as promised by condition `T`.

We now have a list of roots, sorted by time, that can be used to access previous versions of D. For queries, we assume the search starts from the relevant root. Otherwise, the relevant root can be found in $\mathcal{O}(\log r)$ time, where r is the total number of roots. Timestamps on edges represent the versions in which the edge was present, so it is easy to adapt queries to the DAG to only take into

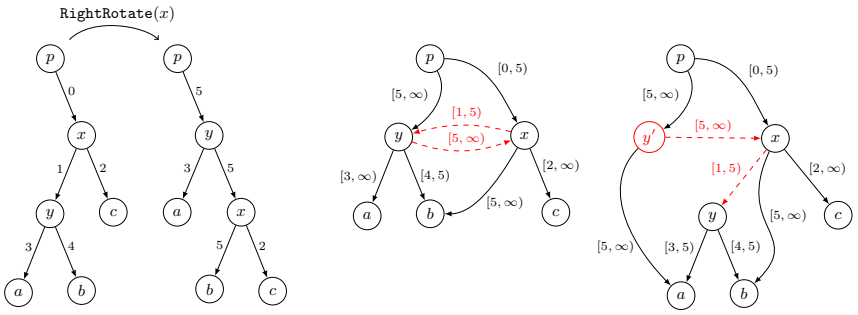


Fig. 3: Left: a rotation in a BST. Middle: the resulting graph contains the dashed cycle between x and y. Right: by creating a new copy y' of y the resulting graph remains a DAG and is TUNA compliant. We use the same notation as in Fig. 1.

account the relevant edges among the $\Theta(d)$ present edges in the node, leading to the same asymptotic running time for queries. □

5 Further applications

Okasaki [18] introduces functional *random access arrays*, achieving, for an array of size n, worst-case lookup and update time $\mathcal{O}(\min\{i, \log n\})$, where i is the index of the queried element. This data structure easily satisfies the TUNA conditions, and can therefore be made offline partially persistent with only a constant factor space overhead.

Condition U of Definition 1 ensures that each of the n updates makes $\mathcal{O}(1)$ edges, which then totals to $\mathcal{O}(n)$ edges in the final structure. In the analysis of the space complexity, it is however not important exactly how many edges each update adds, and in fact, the more general property holds that the space usage is linear in the number of edges created by the ephemeral structure. Condition U can therefore be relaxed to each update producing for example amortized or expected $\mathcal{O}(1)$ new edges. This allows for balanced *Red-Black trees* [5] to fulfill the TUNA conditions even if the colors should be stored for persistent queries, as they make amortized $\mathcal{O}(1)$ color changes for each update [27]. The simpler functional implementation of Red-Black trees by Okasaki [20] makes amortized $\mathcal{O}(1)$ changes per insertion by a similar argument. See Fig. 3 on how rotations can be handled. Similarly, *Treaps* [3] fulfill the relaxed TUNA conditions, as each update makes expected $\mathcal{O}(1)$ rotations and therefore $\mathcal{O}(1)$ new edges.

Condition A ensures that all nodes can be reused and that storing edges is sufficient to produce the query DAG. The underlying tree is always represented explicitly, and as updates always operate on the current structure, it is possible to relax condition A, to allow for dynamic update information in each node. This information can be altered during the update phase and is not needed to produce the query DAG. Recall that our structure imposes additive $\mathcal{O}(n \log n)$

construction time independent of original update complexity. This allows for *AVL-trees* [1] to fulfill the `TUNA` conditions, as the balance value (the height of the subtree rooted at the node) is only used for balancing during the updates, and updates only perform amortized $\mathcal{O}(1)$ rotations [2].

6 Implementation and experiments

The construction described in this chapter has been implemented and tested in Haskell[2]. Experiments were performed on WSL Ubuntu 20.04.6 on Windows 10.0.19044, with Processor 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 4 Core(s), 8 Logical Processor(s), 16 GB RAM, running `ghc` haskell compiler version 8.6.5, without any compiler flags. Each runtime measurement is the average of a constant number of runs on the same input. A random input is generated by providing the algorithm with a seed to a pseudo-random number generator.

For the following, let the version of a data structure made in our offline-partially-persistent framework be denoted as *persistent* and the version made in regular Haskell without necessarily saving the root of the structure be denoted *ephemeral*. The implementation uses time intervals for the edges in the freezer, and not only a start timestamp. Edges in the current structure are therefore not recorded in the freezer before they are removed from the live structure. Therefore, if any larger part of the structure is to be removed, it must be done so recursively, to correctly enter all of the edges into the freezer. Note that this does not alter the amortized running time of updates.

To test the *correctness* of the implementation, we apply various deterministic and random sequences of updates to both an ephemeral and a persistent unbalanced binary search tree, saving the roots of the ephemeral versions. We then construct the query DAG as described in Section 3.2 based on the persistent version. Then, for each timestamp, we tested if both versions produced the same tree. The test did not reveal any errors.

To measure space usage of a data structure, Haskell provides the function `recursiveSizeNF` which recursively measures the size of the object in bytes, i.e., traverses the whole pointer structure on the heap. Note that parts of the structure reachable from multiple places only are counted once. The space of the persistent data structure is measured after building the query DAG. Note that due to [25], a sequence of uniformly random insertions in an unbalanced binary search tree produces expected depth $\Theta(\log n)$.

The experiments for space usage, runtime of updates, and queries follow the expected result from the theory, see Figs. 4 and 5a–d. The experiment for the DAG building runtime appeared to be more than the theoretical $\mathcal{O}(n \log n)$ from the plot, see Fig. 5e. We are unable to find an explanation for the overhead. To ascertain the source of the overhead, we ran a sanity experiment to test if this overhead occurred on simpler problems. We inserted 1 to n into an unbalanced ephemeral BST where the insertion order created an almost perfectly balanced

[2] Available at `https://github.com/Crowton/Persistent-Functional-Trees`.

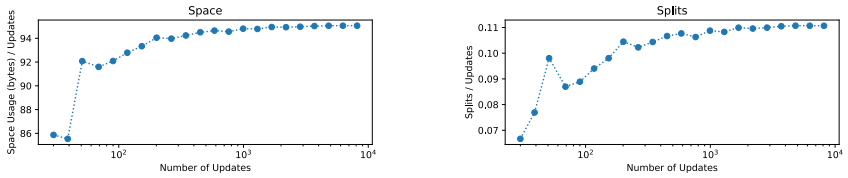


Fig. 4: Space usage experiments. Elements 1 to n are inserted into a persistent BST in order to create a path. Element $n+1$ was then inserted and deleted n times, to introduce cascading node splits up the path, for a total of $3n$ updates.

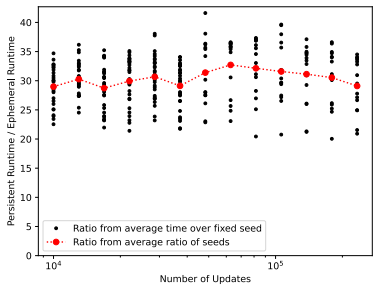
tree, via careful insertion of elements. Then all elements were queried in random order. This simple experiment has the clean theoretical runtime of $\mathcal{O}(n \log n)$ but similarly turned out not to produce an $\mathcal{O}(n \log n)$ plot in practice, see Fig. 5f, leading to the conjecture that the extra overhead in runtime is not from the program itself, but the Haskell compiler, specific implementations of underlying structures, and/or the environment the code is executed in.

The runtime experiments of updates and queries showed no issues with extra logarithmic factors, as only the relative runtime of the ephemeral and persistent implementation is compared. Here we found a constant factor difference, and thus the experiments did not disprove Theorem 1.

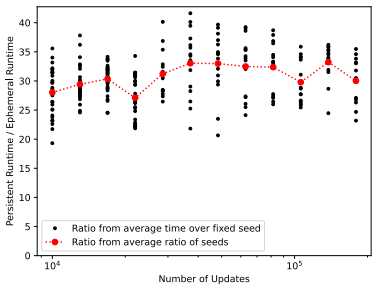
References

1. Adel'son-Vel-skii, G.M., Landis, E.M.: An algorithm for the organization of information. Soviet Math. Doklady **3**, 1259—1263 (1962)
2. Amani, M., Lai, K.A., Tarjan, R.E.: Amortized rotation cost in AVL trees. Inf. Process. Lett. **116**(5), 327–330 (2016). https://doi.org/10.1016/j.ipl.2015.12.009
3. Aragon, C.R., Seidel, R.: Randomized search trees. In: 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989. pp. 540–545. IEEE Computer Society (1989). https://doi.org/10.1109/SFCS.1989.63531
4. Backus, J.: Can programming be liberated from the von neumann style? a functional style and its algebra of programs. Commun. ACM **21**(8), 613—-641 (aug 1978). https://doi.org/10.1145/359576.359579
5. Bayer, R.: Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica **1**, 290–306 (1972). https://doi.org/10.1007/BF00289509
6. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. Journal of Functional Programming **6**(6), 839—-857 (1996). https://doi.org/10.1017/S095679680000201X
7. Buchsbaum, A.L., Tarjan, R.E.: Confluently persistent deques via data-structural bootstrapping. Journal of Algorithms **18**(3), 513–547 (1995). https://doi.org/10.1006/jagm.1995.1020
8. Cole, R.: Searching and storing similar lists. Journal of Algorithms **7**(2), 202–220 (1986). https://doi.org/10.1016/0196-6774(86)90004-0

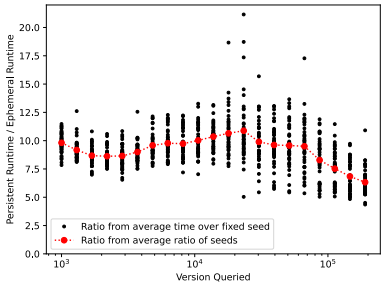
9. Demaine, E.D., Langerman, S., Price, E.: Confluently persistent tries for efficient version control. In: Scandinavian Workshop on Algorithm Theory. pp. 160–172. Springer (2008). https://doi.org/10.1007/s00453-008-9274-z
10. Dobkin, D., Lipton, R.J.: Multidimensional searching problems. SIAM Journal on Computing **5**(2), 181–186 (1976). https://doi.org/10.1137/0205015
11. Dobkin, D.P., Munro, J.I.: Efficient uses of the past. In: Proceedings of the 21st Annual Symposium on Foundations of Computer Science. pp. 200—206. SFCS '80, IEEE Computer Society, USA (1980). https://doi.org/10.1109/SFCS.1980.18
12. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. pp. 109—121. STOC '86, Association for Computing Machinery, New York, NY, USA (1986). https://doi.org/10.1145/12130.12142
13. Edelsbrunner, H., Guibas, L.J., Stolfi, J.: Optimal point location in a monotone subdivision. SIAM Journal on Computing **15**(2), 317–340 (1986). https://doi.org/10.1137/0215023
14. Hughes, J.: Why functional programming matters. In: Turner, D.A. (ed.) Research Topics in Functional Programming. pp. 17–42. The UT Year of Programming Series, Addison-Wesley (1990). https://doi.org/10.1093/comjnl/32.2.98
15. Kahn, A.B.: Topological sorting of large networks. Commun. ACM **5**(11), 558—562 (nov 1962). https://doi.org/10.1145/368996.369025
16. Kaplan, H., Tarjan, R.E.: Persistent lists with catenation via recursive slow-down. In: Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing. pp. 93–102 (1995). https://doi.org/10.1145/225058.225090
17. Kirkpatrick, D.: Optimal search in planar subdivisions. SIAM Journal on Computing **12**(1), 28–35 (1983). https://doi.org/10.1137/0212002
18. Okasaki, C.: Purely functional random-access lists. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture. pp. 86—95. FPCA '95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/224164.224187
19. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, USA (1999)
20. Okasaki, C.: Red-black trees in a functional setting. Journal of Functional Programming **9**(4), 471–477 (1999). https://doi.org/10.1017/s0956796899003494
21. Overmars, M.H.: Searching in the past II: general transforms. Tech. rep., Tech. Rep. RUU (1981)
22. Overmars, M.H.: The Design of Dynamic Data Structures, Lecture Notes in Computer Science, vol. 156. Springer (1983). https://doi.org/10.1007/BFb0014927
23. Pippenger, N.: Pure versus impure lisp. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 104—109. POPL '96, Association for Computing Machinery, New York, NY, USA (1996). https://doi.org/10.1145/237721.237741
24. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. Commun. ACM **29**(7), 669—679 (jul 1986). https://doi.org/10.1145/6138.6151
25. Sedgewick, R., Flajolet, P.: An Introduction to the Analysis of Algorithms. Addison-Wesley Longman Publishing Co., Inc., USA (1996)
26. Seidel, R.: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Computational Geometry **1**(1), 51–64 (1991). https://doi.org/10.1016/0925-7721(91)90012-4
27. Tarjan, R.E.: Amortized computational complexity. SIAM Journal on Algebraic Discrete Methods **6**(2), 306–318 (1985). https://doi.org/10.1137/0606031



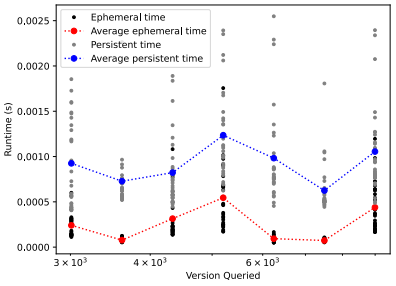
(a) Inserting elements 1 to n in random order.



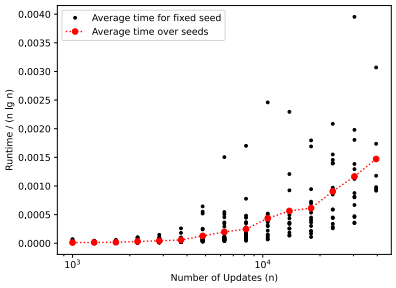
(b) Inserting and deleting elements 1 to n in random order.



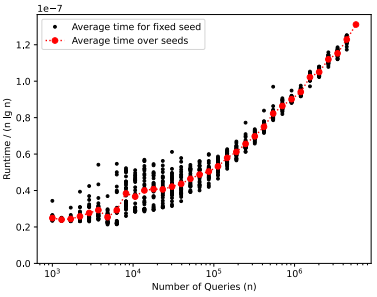
(c) Relative running time over querying all elements of a persistent BST in order at different timestamps. The tree is created by inserting elements 1 to 200000 in random order.



(d) Elements 1 to n are inserted to make a path. Element $n+1$ is inserted and deleted n times, for $n = 3000$. Time measured for querying element $n+1$ at different timestamps.



(e) DAG building running time experiment. Elements 1 to n were inserted and deleted from a persistent BST in random order. Time was measured on the DAG building alone.



(f) Sanity experiment. Querying all elements of an emphemeral perfectly balanced BST of size n in random order.

Fig. 5: Running time experiments.

Chapter 9

Polynomial-Time Algorithms for Contiguous Art Gallery and Related Problems

The Contiguous Art Gallery Problem is Solvable in Polynomial Time

Magnus Christian Ring Merrild
Department of Computer Science, Aarhus University, Denmark

Casper Moldrup Rysgaard
Department of Computer Science, Aarhus University, Denmark

Jens Kristian Refsgaard Schou
Department of Computer Science, Aarhus University, Denmark

Rolf Svenning
Department of Computer Science, Aarhus University, Denmark

Abstract

In this paper, we study the *Contiguous Art Gallery Problem*, introduced by Thomas C. Shermer at the 2024 Canadian Conference on Computational Geometry, a variant of the classical art gallery problem from 1973 by Victor Klee. In the contiguous variant, the input is a simple polygon P, and the goal is to partition the boundary into a minimum number of polygonal chains such that each chain is visible to a guard. We present a polynomial-time RAM algorithm, which solves the contiguous art gallery problem. Our algorithm is simple and practical, and we make a C++ implementation available.

In contrast, many variations of the art gallery problem are at least NP-hard, making the contiguous variant stand out. These include the classical art gallery problem and the *edge-covering problem*, both of which being proven to be $\exists\mathbb{R}$-complete recently by Abrahamsen, Adamaszek, and Miltzow [J. ACM 2022] and Stade [SoCG 2025], respectively. Our algorithm is a greedy algorithm that repeatedly traverses the polygon's boundary. To find an optimal solution, we show that it is sufficient to traverse the polygon polynomially many times, resulting in a runtime of $\mathcal{O}\big(n^6 \log n\big)$ arithmetic operations. We further bound the bit complexity of the computed values, showing that problem is in P. Additionally, we provide algorithms for the restricted settings, where either the endpoints of the polygonal chains or the guards must coincide with the vertices of the polygon.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Art Gallery, Computational Geometry, Combinatorics, Discrete Algorithms

Funding Supported by Independent Research Fund Denmark (DFF), grants 9131-00113B and 10.46540/3103-00334B

Acknowledgements We thank Joseph O'Rourke for organizing the open problem session [4] at the Canadian Conference on Computational Geometry 2024 (CCCG24) at Brock University, Canada, where Thomas C. Shermer proposed the contiguous art gallery problem. The authors also thank the participants of CCCG24 for their lively discussions of the problem at the conference, especially Frederick Stock. This paper was submitted to the Symposium on Computational Geometry 2025 conference and accepted as a merge with [28] and [8] into [7]. We thank our collaborators for the engaging discussions that inspired optimizations to our work.

1 Introduction

The *art gallery problem*, introduced by Victor Klee in 1976, is a classical computational geometry problem where the goal is to find a minimum set of guards (points) in the interior of an input polygon P that sees every other point in P of the polygon. There are numerous variations of this problem, many of which are at least NP-hard or require complicated algorithms with a high polynomial running time. This is particularly the case for unrestricted

Table 1 Summary of the work related to the contiguous art gallery problem, for polygons with n vertices where k^* is the size of the minimal decomposition. We abbreviate Art Gallery as AG, † refers to the guard location being restricted to vertices, and ‡ refers to the vertices of the pieces (the guarded area) being restricted to the vertices of the input polygon. Notice how the difficulty of the four problems increases along the diagonal from the bottom left vertex to the top right.

Problem	Vertex restricted	Unrestricted	With holes
Standard AG	NP-hard† [22]	$\exists\mathbb{R}$-complete [1]	$\exists\mathbb{R}$-complete [1]
Edge-covering AG	NP-hard† [19]	$\exists\mathbb{R}$-complete [31]	$\exists\mathbb{R}$-complete [31]
Star-shaped partition	$\mathcal{O}\big(n^7 \log n\big)^{\ddagger}$ [17]	$\mathcal{O}\big(n^{105}\big)$ [2]	NP-hard [26]
Contiguous AG	$\mathcal{O}\big(n^2 \log^2 n\big)^{\ddagger}$ *new* $\mathcal{O}\big(n^2 \log n\big)^{\dagger}$ *new*	$\mathcal{O}\big(k^* n^5 \log n\big)$ *new*	*Open*

variants, where guard locations and the part of the polygon they cover are not constrained to the vertices of the input polygon. In this paper, we study the *contiguous art gallery problem* where the boundary of P should be partitioned into a minimum number of contiguous intervals, i.e., polygonal chains, such that each chain is visible to a guard in the interior of P. Neither the guards nor the endpoints of the chains are restricted to the vertices of P. The problem was introduced by Thomas C. Shermer at the Canadian Conference on Computational Geometry 2024. We resolve it by providing a polynomial-time real RAM algorithm.

▶ **Theorem 1.** *The contiguous art gallery problem for a simple polygon with n vertices is solvable in $\mathcal{O}\big(k^* n^5 \log n\big)$ arithmetic operations, where k^* is the size of an optimal solution.*

In section 6 we show that the bit complexity of the arithmetic operations are bounded polynomially in the number of bits used to encode the input polygon, therefore showing the aforementioned algorithm is in fact a polynomial-time RAM algorithm.

▶ **Theorem 2.** *The contiguous art gallery problem for a simple polygon encoded in N bits is a member of the complexity class P.*

In contrast to many other art gallery variants, it is a surprising and positive result that this variant allows for an efficient and simple algorithm. We demonstrate this by implementing the algorithm in C++ using CGAL [11, 12, 16, 32, 36]. Our code is available online [25].

1.1 Related work

The literature contains a variety of variations of the classical art gallery formulation [26, 30, 33], and we summarize the most important ones in Table 1. Most art gallery variants can be framed as *decomposition problems* [18, 23], where the goal is to decompose an input polygon P into less complicated components whose union is P. If the pieces may overlap, it is *covering problem*, and if not it is a *partition problem*. In the art gallery setting each piece must be guarded. The variants can further be categorized as *restricted* or *unrestricted*. In a restricted version of the problem, the guards and/or the vertices of each piece must coincide with the vertices of P. Conversely, an unrestricted version imposes no such constraints on the placement of guards or the vertices of the pieces. Variants also differ based on the complexity of P. We consider the two important cases of simple polygons with and without holes, with

the latter typically being significantly more complicated. In Table 1 summarizes the difficulty of three fundamental art gallery variants and our contiguous variant.

The classical art gallery problem can be viewed as a covering problem in which the input polygon is decomposed into a minimal number of *star-shaped* polygons. A polygon is star-shaped exactly when it is possible to place a guard that sees all other points in the polygon. The decision problem *"Can this polygon be guarded by k guards?"* was recently shown by Abrahamsen, Adamaszek, and Miltzow [1] to be $\exists\mathbb{R}$-complete [29] for any simple polygon with or without holes. If the guards are restricted to the vertices of the polygon, then Lee and Lin [21] showed that the problem is NP-hard.

The edge-covering problem is a variant of the art gallery problem, where the aim is to only cover the edges of a polygon, motivated by protecting valuable art on the walls of the gallery. The edge-covering variant was shown by Laurentini to be NP-hard [19] for guards restricted to vertices and $\exists\mathbb{R}-$hard for unrestricted guards and polygons with holes. As seen in Figure 3, covering the edges does not imply covering the interior of P.

The minimum star-shaped partition problem was shown by Keil [17] to be solvable in $\mathcal{O}(n^7 \log n)$ arithmetic operations, when the star-shaped regions start and end at vertices. Without vertex restriction, Abrahamsen, Blikstad, Nusse, and Zhang [2] introduced a breakthrough algorithm solving this harder variant using $\mathcal{O}(n^{105})$ arithmetic operations. With holes, computing the minimum star-shaped partition is NP-hard due to O'Rourke [26].

The contiguous art gallery problem was introduced at the 2024 Canadian Conference on Computational Geometry, by Thomas C. Shermer and this variant is the study of this paper. Framed as a decomposition problem, the goal is to partition the boundary ∂P into a minimal number of polygonal chains such that each can be seen by a guard interior to P. We mainly focus on the unrestricted version for a simple polygon without holes, but we also describe algorithms for restricted versions. Algorithm 1 does not generalize to polygons with holes, see Appendix C.2, and determining the hardness of this variant is an interesting open problem. The placement of guards in optimal solutions may differ across all variants, as demonstrated in Figures 3 and 4.

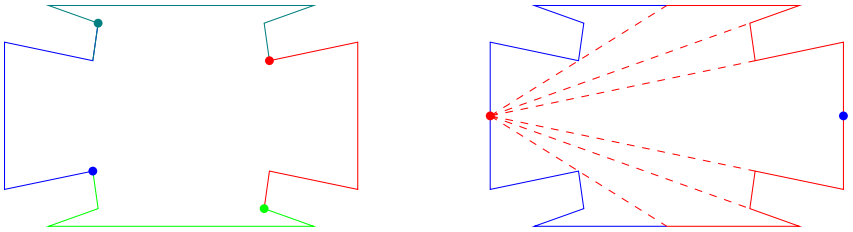


Figure 1 Left, an optimal solution to the vertex restricted guards contiguous art gallery problem, discovered by greedily maximizing in one direction. Right, an optimal unrestricted contiguous art gallery solution requires 2 guards whose guarded pieces start and end at non-vertex points on the boundary, and these boundary points are not directly defined by segments of the polygon.

1.2 Limitations of Existing Approaches

We first sketch why the restricted version of the contiguous art gallery problem is relatively simpler to solve in polynomial time than the unrestricted variant, which is our focus. When partitioning the boundary of a simple polygon, it is natural to view it as a circle and the polygonal chains as arches or intervals; see Figure 12. By doing so, finding a minimal partition is similar to finding a *minimal circle-cover* among a set of intervals C. This problem

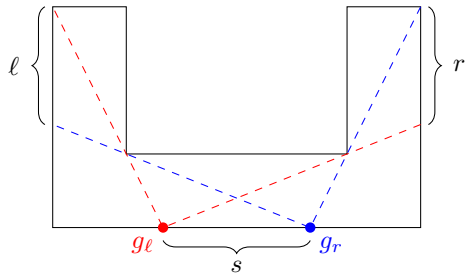


Figure 2 A polygon demonstrating where the blue guard g_r sees all of r and none of ℓ, and the red guard g_ℓ is symmetric. Any guard placed along s will yield a trade-off between how much of ℓ and r it sees. This shows that it is non-trivial with unrestricted guards to find a finite set of polygonal chains that include an optimal solution as there may be infinite maximal intervals.

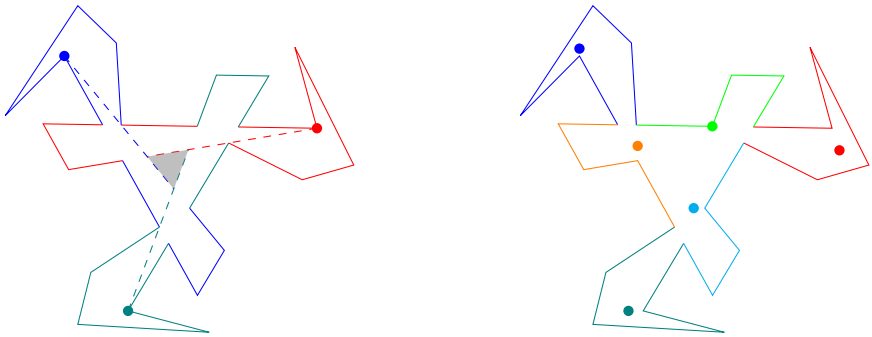


Figure 3 Left, 3 optimal edge-covering guards where the inner gray triangle is not visible. Right, 6 guards that optimally contiguously cover the boundary. Guards are visualized as points whose color matches the part of the boundary they guard.

was studied by Lee and Lee [20] for finite cardinality C, giving an algorithm that runs in $\mathcal{O}(|C| \log |C|)$ time. Thus, given a finite set C of polygon chains of the boundary of P such that an optimal solution is contained in C, then the contiguous art gallery problem can be solved in $\mathcal{O}(|C| \log |C|)$ time by simply viewing each polygonal chain as an arc and running the minimal circle-covering algorithm. In the restricted contiguous art gallery problem, that is, restricting guard positions to vertices of P or restricting the endpoints of the polygonal chains to coincide with vertices of P, leads to a set of polygonal chains C that includes an optimal solution of size $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$, respectively. The set C must also be computed, and in Appendix A we describe how to solve these two problems in $\mathcal{O}(n^2 \log n)$ and $\mathcal{O}(n^2 \log^2 n)$ time, respectively.

The unrestricted contiguous art gallery problem is our main focus, where guards may be placed anywhere interior to P, and polygonal chains of the boundary of P may start and end anywhere. In this setting, it is non-trivial to generate a polynomial-sized set of intervals that contains an optimal solution, which would allow the use of the algorithm for the circle-cover minimization problem. An approach that does not work is to generate all the different candidate intervals that are maximal, as there may be infinitely many of these, as shown in Figure 2. Another approach that we were unable to rule out is based on showing that an optimal solution coincides with a point of low *degree* [2] either with a guard or the endpoint of a polygonal chain. Points of degree 0 are the vertices of P, and points of degree $i > 0$ are points formed by intersections of lines formed by pairs of points of degree $(i - 1)$.

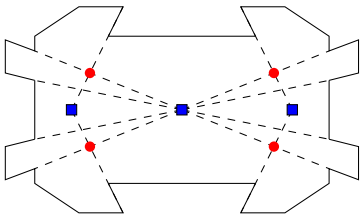


Figure 4 A polygon, where the red guards (circles) represent an optimal solution to the contiguous art gallery problem and the blue guards (squares) are an optimal solution to the classical art gallery problem.

Letting D_i be the number of points of degree i, then D grows as $\Theta\left(n^{4^i}\right)$ so even if this approach is feasible, it leads to high polynomial running time even for low degree points. For the minimum star-shaped partition problem the authors showed that first-degree points suffice as candidates [2].

1.3 Our Greedy Optimal Solution

Our solution takes a different and remarkably simple approach. At the core of our method is the GreedyInterval algorithm, which, given a point x on the polygon's boundary, finds the furthest point y along the boundary in the clockwise direction such that the polygonal chain from x to y can be guarded by an interior point of the polygon. We denote these as *greedy intervals*. Repeatedly taking these greedy steps until the boundary has been covered gives a solution of size $k \leq k^* + 1$. Each traversal of the boundary we denote by a *revolution*, and continuing this process performing $\mathcal{O}(kn^3)$ revolutions ultimately yields an optimal solution. Performing a single revolution by repeatedly using the GreedyInterval algorithm takes $\mathcal{O}(n^2 \log n)$ arithmetic operations, leading to $\mathcal{O}(k^* n^5 \log n)$ total arithmetic operations. The model of computation used here is the real RAM model [9] where arithmetic operations on real numbers take unit time. Here, the size of the input n is the number of real-valued inputs, that is, the vertices of a circular list of vertices of P. The main result of our paper is Theorem 1 that the contiguous art gallery problem is solvable in polynomial time in the real RAM model.

We also consider the problem in the RAM model, where the polygon is represented as an array of points, each point having two coordinates consisting of fractions with integer numerator and denominator. We consider the case where N bits are used to describe the entire input and show that $poly(N)$ operations is sufficient to find an optimal guarding. This is the contents of Section 6.

When the input polygon has holes our approach does not work and it is an intriguing open problem to determine whether this variant of the contiguous art gallery problem is solvable in polynomial time.

1.4 Concurrent work

This work is concurrent with [28] and [8] that both solve the contiguous art gallery problem using different techniques, the merge of this paper [7] was published at SoCG25. A notable difference is that this paper traverses the polygon clockwise, while the three others are counter-clockwise.

In [28] it is explored how GreedyInterval can be expressed as a piecewise rational linear function and show that repeatedly composing GreedyInterval with itself leads to a

piecewise rational linear function representing *all* optimal solutions.

In [8] it is explored how to generate $\mathcal{O}(n^4)$ candidate solutions by starting Algorithm 1 from a carefully selected set of points, such that at least one of these will be optimal.

1.5 Organization

In Section 2 we present our algorithm for solving the contiguous art gallery problem, and in Section 3 we cover details of how our algorithm can be implemented using basic computational geometry operations. In Sections 4 and 5, we derive an upper bound on the number of iterations of our greedy algorithm based on the geometric and combinatorial properties of our algorithm. In Section 6 we then use this bound to prove that the contiguous art gallery problem is in the complexity class P. Finally, in Section 7 we state related open problems.

In Appendix A we describe how to solve the contiguous art gallery problem under vertex restrictions. In Appendix B we prove geometric supporting lemmas. In Appendix C we give two examples that illustrate interesting behavior of Algorithm 1.

2 The Repeated Greedy Algorithm

Formally, let P be a simple polygon with n vertices $v_0, \ldots, v_{n-1}$ and edges e_i connecting v_i and v_{i+1}. The boundary of P we denote as ∂P. The goal of the contiguous art gallery problem is to find the minimum k^* such that one can partition the boundary ∂P into k^* contiguous *visible* polygonal chains $I_1, I_2, I_3, \ldots, I_{k^*}$ whose union is ∂P. A chain I_i is visible if there exists a guard g_i such that, for all points x on I_i, the line segment $\overline{x\,g_i}$ is contained in P. Conceptually, each I_i corresponds to an interval of the boundary between two points a and b, denoted as $[a, b]$. We allow guards to be collocated and consider them non-blocking, i.e., they can see through each other. Furthermore, for convenience, we always index the edges and vertices of P modulo n, meaning that $e_n = e_0$ and $v_n = v_0$.

Algorithm 1

Input: A simple polygon P with vertices $v_0, v_1, v_2, \ldots, v_n$
Output: An explicit solution to the contiguous art gallery problem on P

1 $x_0 \longleftarrow v_0$
2 **for** $i = 1$ **to** T **do**
3 $\quad x_i, g_i \longleftarrow GreedyInterval(x_{i-1}, P)$
4 $j \longleftarrow \max\{j \leq T-2 \,|\, x_j \in (x_{T-1}, x_T]\}$
5 **return** $\{(x_{i-1}, x_i, g_i) \,|\, j < i \leq T\}$ `// The last segments/guards covering` ∂P

To solve the contiguous art gallery problem, we propose Algorithm 1 which is a conceptually simple greedy algorithm that starts at any point x_0 in ∂P (Line 1). Then repeatedly finds *greedy* intervals, that is, the longest visible interval of ∂P from a given starting point of ∂P, starting from where the previous segment ended (Lines 2 - 3). The greedy interval from any point $x \in P$ can be found in $\mathcal{O}(poly(n))$ time by combining basic computational geometry operations to calculate visibility polygons [5, 10], intersections of polygons [35] and common tangents between disjoint polygons [3]. The GreedyInterval algorithm is described in details in Section 3 as Algorithm 2. After $T \geq ck^2n^3$ iterations, where c is a sufficiently large constant and k is the number of iterations in the first revolution, the algorithm returns the start and endpoint of the last greedy intervals that form a partition of the boundary and the corresponding guards that see the segments (Lines 4 - 5).

3 The GreedyInterval algorithm

In this section, we present and analyze the GreedyInterval algorithm for computing greedy intervals, i.e. the longest visible interval I from a given point $x \in P$, along with a corresponding guard g that sees I. The algorithm uses only basic computational geometry operations. We assume for simplicity that P is not star-shaped, as otherwise $I = \partial P$. The idea is to extend I from one vertex of P to the next (clockwise around ∂P), starting from the edge containing x, while maintaining the feasible region of I. The feasible region of I is the set of points of P that can see all of I, i.e. the region of P where a guard for I can be placed. This happens on lines 2 - 3 where $VP(x, P)$ is a subroutine to compute the *visibility polygon* [10] of point $x \in P$ and $\cap$ computes the intersection between two simple polygons [35]. The correctness of extending I in discrete steps around ∂P follows by contrapositive of Lemma 55, which is that any point $u \in P$ that can see two points a, b on some edge e_j, can see all points on e_j between a and b. Finally, we extend I along edge e_{i-1} towards the last vertex v_i that causes the feasible region to be empty. This happens on line 4 where $lastVisiblePoint(P, F_{i-1}, e_{i-1})$ is a subroutine for computing the point furthest along e_{i-1} visible from F_{i-1}. To do so simply and efficiently we prove Lemma 5 which shows that it suffices to consider the vertices of the last feasible region.

Algorithm 2 GreedyInterval

Input: Point x on edge $e_i = (v_i, v_{i+1})$ of a simple and a non-star-shaped polygon P
Output: The endpoint of the longest visible interval I from x and a guard g that can see I

1 $F_i \longleftarrow VP(P, x)$
2 **while** $F \neq \emptyset$ **do**
3 $\quad i \longleftarrow i + 1$
4 $\quad F_i \longleftarrow F_{i-1} \cap VP(P, v_i)$
5 **return** $lastVisiblePoint(P, F_{i-1}, e_{i-1})$

The main concern is to show that the running time of GreedyInterval is $\mathcal{O}(poly(n))$. Theorem 3 gives the precise output-sensitive running time, depending on the number e of edges of P that are intersected by I.

▶ **Theorem 3.** *The running time of the* GreedyInterval *algorithm for a polygon P with n vertices where the greedy interval that is output intersects e edges of P is $\mathcal{O}(en \log n)$.*

In the following three sections, we clarify how to perform the necessary computational geometry primitives and their efficiency to establish Theorem 3.

3.1 Visibility polygons

For a simple polygon P with n vertices, the *visibility polygon* [10] $VP(P, x)$ of point $x \in P$ is a polygon containing all points of P visible from x. A point that guards x must be placed in $VP(P, x)$, which we also call the feasible region of x. The visibility polygon $VP(P, x)$ has $\mathcal{O}(n)$ vertices and can be computed in $\mathcal{O}(n)$ time using the algorithm by Gindy and Avis [10].

3.2 Intersecting polygons

Computing the intersection of two polygons P and Q with n combined vertices is known as *polygon clipping*, and is a classical problem in computer graphics and computational geometry.

The running time of many of these algorithms, and in particular the output, depends on the number of intersections h between P and Q. An algorithm by Martínez, Rueda, and Feito [24] shows that the problem of polygon clipping can be solved in $\mathcal{O}((n+h)\log n)$ time. Their approach is similar to the classical sweep line algorithm by Bentley and Ottmann [6] for computing the intersections between a set of segments. There are also earlier algorithms for polygon clipping, but they are less efficient or do not handle degenerate cases [14, 34, 35].

In general, the intersection of two polygons of sizes n and m may have size $h = \Omega(nm)$, which could cause the repeated intersections of GreedyInterval to grow to size $\Omega(n^n)$. However, we prove that any feasible region will have at most $\mathcal{O}(n)$ vertices:

▶ **Lemma 4.** *The feasible region of a set of points S in a simple polygon P with n vertices has $\mathcal{O}(n)$ vertices.*

Proof. Let $v_1, v_2, \ldots, v_t$ be the vertices of S. Let $F := VP(P,S)$ be the intersection $\bigcap_{i=1}^{t} VP(P, v_i)$. The boundaries of $VP(P, v_i)$ consists of sub line segments of ∂P and sub line segments of rays from v_i to reflex vertices of P. Hence, the boundary of F consists of the same sub line segments.

Any edge of ∂P can only contribute to one edge of ∂F (since $F \cap C$ is convex for every convex subset of P, Lemma 39), so at most n edges of ∂F come from these. Thus, we only need to restrict the number of edges coming from rays.

Consider a reflex vertex v_i and the rays stemming from visibility polygons defined by v_i. We now consider three rays, r_1, r_2 and r_3, which all pass through v_i. We will show, that at most two of these rays will contain sub line segments that are part of ∂F.

Order the rays, by the angle they make with edge $v_{i-1}v_i$ and assume that r_1 and r_3 contain sub line segments part of ∂F. Thus F will be contained in the half planes consisting of everything away from the path $v_{i-1}v_iv_{i+1}$ on the other side of r_1 and r_3 extended to lines. However, the only intersection of r_2 with these two half planes is v_i, hence it can not contribute an edge to ∂F. Doing this repeatedly, we get that at most two rays through v_i can contribute edges to the feasible region, see Figure 5.

By Lemma 39 it holds that $F \cap C$ is convex for each convex subset C of P, and we get that each ray can contribute at most one edge to the feasible region. Thus each reflex vertex can contribute at most two edges to F, meaning that the complexity of F is at most $3n = \mathcal{O}(n)$.

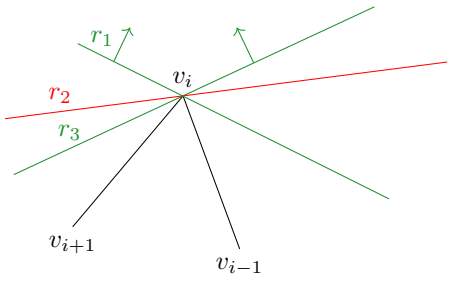


Figure 5 r_1, r_2 and r_3 are rays passing through v_i. If both r_1 and r_3 contribute to the edges of F, then F lies in the quarter plane shown with arrows. Now r_2 will not be able to contribute to edges of F.

◀

Combining Lemma 4 with the polygon clipping algorithm, we get a running time of $\mathcal{O}(n \log n)$ to compute the intersection between the previous feasible region and a visibility polygon.

3.3 The last visible point

When computing $lastVisiblePoint(P, F_{i-1}, e_{i-1})$, the input is a simple polygon P with n vertices, an edge $e_{i-1} = (v_{i-1}, v_i)$, and a feasible region F_{i-1} with f vertices where v_{i-1} is visible from all points of F_{i-1} and v_i is not visible from any point of F_{i-1}. The output is the point y on e_{i-1} closest to v_i visible from F_{i-1} and the guard $g \in F_{i-1}$ that can see y. We first show that it suffices to consider only the vertices of F_{i-1}.

▶ **Lemma 5.** *Let P, F_{i-1}, and e_{i-1} be defined as above. Then the vertices of F_{i-1} can see at least as far along e_{i-1} as all of F_{i-1}.*

Proof. We prove the lemma by showing that points in the interior of F_{i-1} can see no further than points on the boundary ∂F_{i-1} which again can see no further than the vertices of F_{i-1}.

First, let y_I be a point on e_{i-1} visible from a point p_I in the interior. Clearly, the point of $\overline{p_I\, y_I} \cap F_{i-1}$ with minimal distance to y_I is a point on the boundary of F_{i-1} that can see y_I.

Second, let y_B be a point along e_{i-1} visible from a point p_B on edge (a, b) of the boundary of F_{i-1} that can see y_B. Note F_{i-1} cannot intersect the line $H := L(v_{i-1}, v_i)$ defined by e_{i-1}, since that would imply a point in F_{i-1} that can see v_i contradicting the input assumption that e_{i-1} cannot be seen from F_{i-1}. Recall that F_{i-1} is connected by Lemma 39 which further implies that it must lie exclusively on one side of H. If F_{i-1} is below H then all points of F_{i-1} can see no further than v_{i-1}. Otherwise, F_{i-1} is above H, and in particular edge (a, b) of F_{i-1} is above H. Let a (symmetrically b) be the vertex on the same side of the half-plane defined by $L(p_B, y_B)$ as v_{i-1}. By the contrapositive of Lemma 55 the triangle $\triangle p_B y_B v_{i-1}$ is inside P. Thus, if a is in $\triangle y_B p_B v_{i-1}$ it can also see y_B. Otherwise, a and v_i lie on opposite sides of the half-plane defined by (p_B, v_{i-1}). See Figure 6 for an example of this case. Since a can see v_{i-1}, the triangle $\triangle a p_B v_{i-1}$ is also inside P by the contrapositive of Lemma 55. Thus, the segment from a to y_B is fully contained in triangles $\triangle a p_B v_{i-1}$ and $\triangle y_B p_B v_{i-1}$, which implies that a can see y_B. ◀

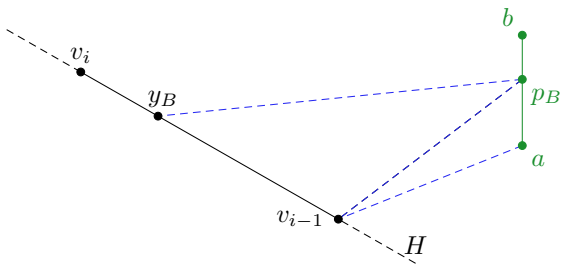


Figure 6 The case in the proof of Lemma 5 where vertex a from the feasible region F_{i-1} satisfies the following conditions: (1) it lies above the half-plane H (2) it is on the same side of the half-plane defined by (p_B, y_B) as v_{i-1} (3) it is on the opposite side of the half-plane defined by (p_B, v_{i-1}) from v_i.

▶ Remark 6. The proof of Lemma 5 also shows that if multiple guard placements are optimal, they must be collinear with the endpoint (y_B above).

Thus we want to find the vertex of F_{i-1}, which sees most of e_{i-1}. We do this by computing the common tangents with a part of ∂P and F_{i-1}. First, we need some terminology:

▶ **Definition 7** (Free area and congested area). *Consider $e_{i-1} = (v_{i-1}, v_i)$ and F_{i-1}. Let t_r and t_m be the right and left tangent to F_{i-1} going through v_{i-1}. The region between these two tangents bounded by F_{i-1} will be without any part of ∂P (since F_{i-1} can see v_{i-1}) and we will therefor call it the free area. Let the left tangent to F_{i-1} through v_i be t_ℓ. The region bounded by t_m, t_ℓ and e_{i-1} will need to contain parts of ∂P and will be called the congested area. Let the intersection point between t_ℓ and t_m be q, see Figure 7.*

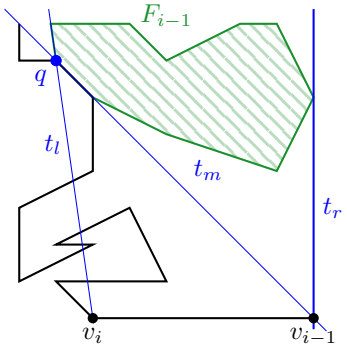


Figure 7 The last feasible region marked in green and tangents t_r, t_m and t_ℓ marked in blue.

The only parts of ∂P, that can block F_{i-1} from seeing v_i is the parts in the congested area. Thus, we construct a polygon from the ∂P in the congested area:

▶ **Definition 8** (Blocking polygon). *We define the blocking polygon B by considering the following path: Starting from v_i, follow t_ℓ until you hit ∂P again. Then follow ∂P into the congested region until you hit t_ℓ again. Continue this until you hit q or F_{i-1}. If one hits q, one moves a little to the left to a point r that lies below t_m and then finishes by moving parallel to t_ℓ until you can move directly left to v_i.*

If you hit F_{i-1} before q, then this must occur along t_m. By Lemma 39, F_{i-1} will only touch t_m along one edge of ∂F_{i-1}, so this edge is the one our path will touch. When the path hits this edge, we back up a small bit (small here depends on how close the rest of the path previously has gone to t_m and the distance from the point of contact to q) and then head directly to r (defined just as above) and then finish as above. By choosing the distance of the backtrack and distance between q and r small enough, it can be made so that the path does not intersect itself, as the path could not have hit t_m before the intersection point with F_{i-1}. This is illustrated on Figure 8. This path will be ∂B and B is the region bounded by this path.

Now the claim is the following: F_{i-1} and B are disjoint polygons and one of the common tangents between F_{i_1} and B will be the line of sight of the guard that sees the most of e_{i-1}, see Figure 9. Thus, to find $lastVisiblePoint(P, F_{i-1}, e_{i-1})$ we can run the CommonTangents algorithm of Abrahamsen and Walczak [3] to find all common tangents (of which there are at most four) and manually check them to find the guard position that sees the most of e_{i-1}. The claims will be proven in the following.

▶ **Lemma 9** (Disjoint). *B and F_{i-1} are disjoint.*

Proof. F_{i-1} is contained in the free area and B is contained below t_m. Thus, the only intersection points will be along t_m. By backtracking on the path of ∂B before an intersection

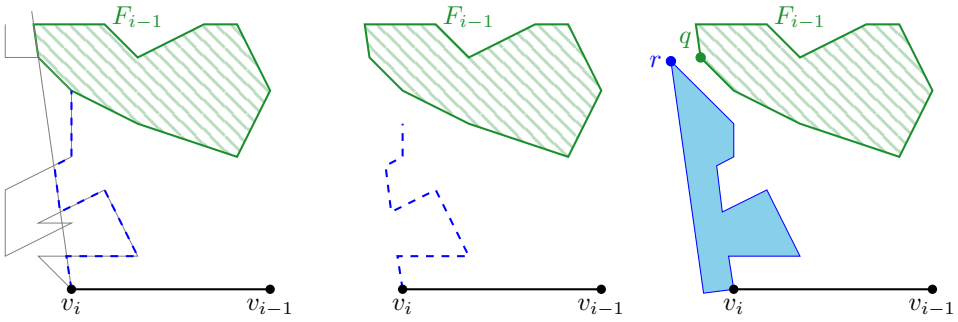


Figure 8 Following t_ℓ and ∂P into the congested area, one traces the boundary of B, marked in blue. Once we hit F_{i-1} we backtrack and head directly for r.

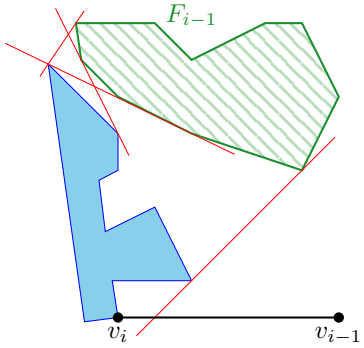


Figure 9 Common tangents drawn. One of which sees the line of sight from the optimal guard to the farthest point on e_{i-1}, which is visible from F_{i-1}

with F_{i-1} and then going directly to r, we no longer touch the free area for the rest of the path, so ∂B does not touch F_{i-1} and must B not either. ◀

Let g be the vertex of F_{i-1} that sees the most of e_{i-1} and let c be the first vertex of ∂P blocking g from seeing any more of e_{i-1} (when moving along ∂P from v_i clockwise). We want to show that $L(g, c)$ is a common tangent of F_{i-1} and B. First, we show that c is a vertex of B.

▶ **Lemma 10** (c is present). *c is a vertex of B.*

Proof. For c to be the blocking vertex, it must be in the congested area. Many vertices of P, which are in the congested area, are vertices of B. The only ones that get skipped are ones that are hidden by other parts of ∂P like vertex h in Figure 10, and vertices that would be skipped, because we hit F_{i-1} and head directly for r.

Vertices such as h cannot be the blocking vertex, since to draw a line from F_{i-1} to h, one must cross other edges of ∂P, so we exclude these.

Assume now that the path followed in Definition 8 where to intersect F_{i-1}, and the first time this happens is at point s. If s itself is a vertex of ∂P, it cannot be c, as by placing a guard at s (which is in F_{i-1}), one sees past s and sees more of e_{i-1} than g would, contradicting that s blocked the guard that sees the farthest.

Now assume that c is a vertex of ∂P in the congested area, which would have been hit after s by following ∂P and t_ℓ as in Definition 8 if we did not skip past them after hitting

s. Since the part of ∂P, in the congested region that contains s must enter and leave the congested area via t_ℓ, the path after s must continue left towards t_ℓ or stay away from t_ℓ. So c would be stuck to the left of the part of ∂B until s. So we again have a problem, since drawing a line from F_{i-1} to s, will hit ∂B in the congested area, i.e., hit ∂P and c cannot be a blocking vertex for g.

Since none of the above cases for c are possible, it must be that c is a vertex of B.

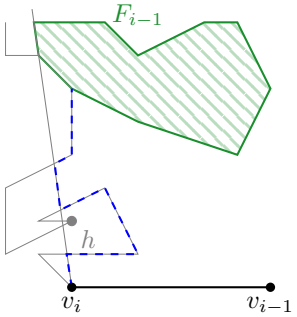


Figure 10 h is hidden behind other parts of ∂P, so h cannot be a blocking vertex.

◀

Finally, we show that $L(g, c)$ is a common tangent.

▶ **Lemma 11** (Sight line is a common tangent). *$L(g, c)$ is a common tangent of F_{i-1} and B*

Proof. Consider the area below $L(g, c) =: \ell$ let ℓ intersect e_{i-1} at y, see Figure 11. If a point g' of F_{i-1} lies below ℓ, g' will see farther along e_{i-1} than y: The line segment $\overline{g'\,y}$ will now not touch B, as it is strictly below ℓ. Thus there will be some point y' further along e_{i-1} which also is not blocked from view of g'. This contradicts the optimality of g's sight, so this cannot happen.

If a point b of B lies below ℓ, it must be a point along ∂P, and this will break the line of sight from g to y, since b must be connected to the rest of B, which lies above ℓ. Thus ℓ is a tangent of F_{i-1} and B.

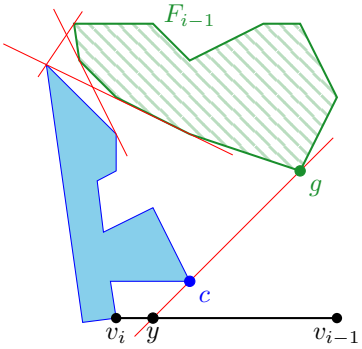


Figure 11 The line of sight, is included in one of the common tangents.

◀

▶ **Proposition 12** (Runtime). *$lastVisiblePoint(P, F_{i-1}, e_{i-1})$ runs in time $\mathcal{O}(n \log n)$*

Proof. $lastVisiblePoint(P, F_{i-1}, e_{i-1})$ has three steps:

1. Compute B.
2. Compute common tangents.
3. Check which tangent is $L(g, c)$ and compute y.

Step 2 is simply running the CommonTangents algorithm of Abrahamsen and Walczak [3], which takes $\mathcal{O}(n)$ time, and Step 3 can also be done in linear time, since there are at most four tangents to check. Thus, we need to show that B can be computed in $\mathcal{O}(n \log n)$ time.

First, we compute t_m and t_ℓ, which can be done in linear time. Next, we compute which segments of ∂P intersect t_ℓ and sort them along t_ℓ. This takes $\mathcal{O}(n \log n)$ time. Finally, determine the vertex or edge of F_{i-1} that lies on t_m. This can be done in linear time.

Now we can trace the path from Definition 8, keeping track of the minimal distance from the vertices considered to t_m. If ∂B does not hit F_{i-1} before reaching q, finish directly. Otherwise, compute a suitable backtrack distance and distance to r, so as not to overlap ∂B. This can be done in constant time, when the distance from ∂B to t_m is known. Then connect to r and finish as before. This takes $\mathcal{O}(n)$ time.

In total, it takes $\mathcal{O}(n \log n)$ time to compute B which is the bottleneck of the algorithm. ◀

3.4 Proving the runtime of GreedyInterval

We now have the tools necessary to prove Theorem 3, as restated below.

▶ **Theorem 3.** *The running time of the* GreedyInterval *algorithm for a polygon P with n vertices where the greedy interval that is output intersects e edges of P is $\mathcal{O}(en \log n)$.*

Proof. Computing the initial feasible region on line 2 takes $\mathcal{O}(n)$ time per Section 3.1. The while loop runs for $e+1 > 2$ iterations and in each iteration, a visibility polygon of complexity $\mathcal{O}(n)$ is computed and the intersection of two polygons of complexity $\mathcal{O}(n)$ is computed, resulting in a new complexity $\mathcal{O}(n)$ polygon (by Lemma 4 the feasible region has complexity $\mathcal{O}(n)$), taking $\mathcal{O}(n \log n)$ operations, according to Section 3.2. Thus, the while loop takes $\mathcal{O}(en \log n)$ operations.

Computing how far the greedy interval extends on the last edge takes $\mathcal{O}(n \log n)$ time per Section 3.3. In total, it takes $\mathcal{O}(en \log n)$ operations to run GreedyInterval. ◀

The above also implies a straightforward bound on the time for Algorithm 1 to perform a revolution of P.

▶ **Corollary 13.** *Repeatedly using the* GreedyInterval *algorithm to perform a revolution of P takes $\mathcal{O}(n^2 \log n)$ time.*

Proof. Let k be the number of iterations of the GreedyInterval algorithm to perform a revolution of P. The bound on the running time follows by applying Theorem 3 to each of the k iterations and using that the number of edges intersected by the greedy intervals sum to at most $2n$. ◀

With the time complexity of GreedyInterval being polynomially bound, we only need to show that a polynomial number of revolutions is needed to find an optimal solution. This will be the content of Sections 4 and 5.

4 Algorithm 1, a combinatorial viewpoint

Summing up what we have done so far: Given any starting point x, the GreedyInterval algorithm finds the point on the boundary y that is furthest from x in the clockwise direction, such that there is a guard that can see the entire polygon chain between x and y. We use the notation $[x, y]$ for this polygonal chain, and call it a *greedy interval*. Generally, for any $a, b \in \partial P$ we let $[a, b]$ denote the polygon chain of ∂P from a to b including a and b. Likewise, we define (a, b) to be the polygon chain of ∂P from a to b excluding a and b. The notations $(a, b]$ and $[a, b)$ are defined analogously, with the inclusion of b or a, respectively. In this section, we focus solely on the combinatorial properties of greedy intervals, i.e. their relative positions to one another. As such, we visualize ∂P as a circle, as shown in Figure 12. In this way, we cast the problem as a circle-cover minimization problem with infinitely many circular arches. The setting with finitely many arches was studied by Lee and Lee [20]. Naturally, some concepts from their work are related, and we will highlight these connections where relevant.

▶ **Definition 14** (Visible intervals). *We denote a polygonal chain of the boundary between vertices a and b as the* interval $[a, b] \subseteq \partial P$. *If one can place a guard in P, that can see all of $[a, b]$ we call it a* visible interval. *The set of all visible intervals is denoted $\mathcal{V}$.*

▶ Remark 15. To simplify the notation and figures, we write G instead of GreedyInterval and let the input polygon and the output guard position be implicit, i.e. G takes a point c on the boundary and returns the farthest clockwise boundary point d such that $[c, d]$ is a visible interval.

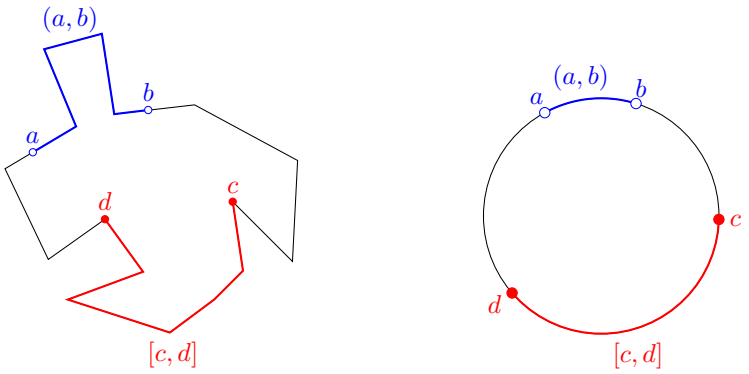


Figure 12 A simple polygon visualized as a circle such that contiguous polygonal chains of the boundary overlap in the polygon if and only if the corresponding intervals overlap in the circle representation. The interval (a, b) is not a greedy interval since b can be moved further around the polygon (clockwise) and remain visible to a guard. The interval $[c, d]$ is a greedy interval since d can be moved no further, i.e. $\mathrm{G}(c) = d$.

▶ Remark 16. We will from now on always assume that P is not star-shaped, as otherwise when we run the GreedyInterval algorithm it will find that the entire boundary can be covered by a single guard, which is clearly optimal. Thus, $\mathrm{G}(x) \neq x$ for all $x \in \partial P$. Furthermore, the greedy interval $[x, \mathrm{G}(x)]$ from any point $x \in \partial P$ always contains at least one edge, since a guard can be placed at the first vertex encountered from x.

▶ **Lemma 17** (Properties of visible intervals).

1. *If* $[a, b] \subseteq [c, d]$ *and* $[c, d] \in \mathcal{V}$ *then* $[a, b] \in \mathcal{V}$.
2. $[x, G(x)] \in \mathcal{V}$.
3. $[x, y] \subseteq [x, G(x)]$ *if and only if* $[x, y] \in \mathcal{V}$.

Proof. 1. and 2. follow by definition. 3. The only if implication follows directly from 1. and 2. For the if implication let $[x, y] \in \mathcal{V}$ and assume for contradiction that $[x, y] \not\subseteq [x, \mathrm{G}(x)]$. Then $\mathrm{G}(x)$ must be strictly before y, contradicting the maximality of GreedyInterval. ◀

Lemma 17 together with Remark 16 acts as combinatorial axioms for our problem. Thus, for the rest of the section, we will use these to obtain properties of GreedyInterval.

▶ **Lemma 18.** *Let* $[a, b] \in \mathcal{V}$ *and* $x, y \in \partial P$*, then*
1. $[x, G(x)] \subseteq [a, b]$ *implies* $G(x) = b$.
2. $x \in [a, b]$ *implies* $b \in [x, G(x)]$.
3. *If* $[x, y] \subseteq [a, b]$*, then* $G(x) \in [b, G(y)]$.

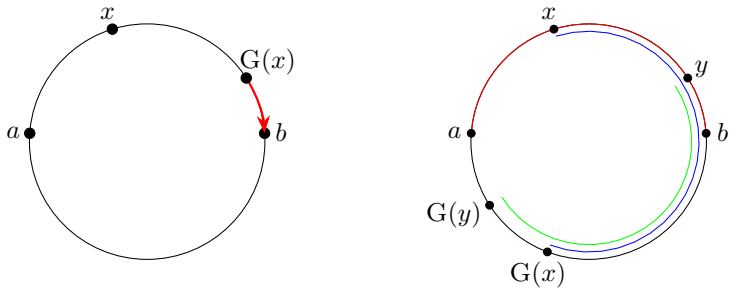


Figure 13 Left, in Lemma 18.1 $\mathrm{G}(x)$ has to reach b or further. Right, in Lemma 18.3 the intervals $[a, b], [x, \mathrm{G}(x)]$ and $[y, \mathrm{G}(y)]$ and their relative positions are marked.

Proof. 1. If $[x, \mathrm{G}(x)] \subseteq [a, b]$ then $[x, \mathrm{G}(x)] \subseteq [x, b] \in \mathcal{V}$, since $[x, b] \subseteq [a, b]$ and $[a, b] \in \mathcal{V}$. Using Lemma 17.3. we get $[x, b] \subseteq [x, \mathrm{G}(x)]$ thus $\mathrm{G}(x) = b$.

2. If $\mathrm{G}(x) \notin [x, b]$, then x, b and $\mathrm{G}(x)$ appear in that order on ∂P, thus $b \in [x, \mathrm{G}(x)]$. If $\mathrm{G}(x) \in [x, b]$ then $\mathrm{G}(x) = b$ by Lemma 18.1.

3. Since $x \in [x, y] \subseteq [a, b] \in \mathcal{V}$ we have by Lemma 18.2. that $y \in [x, \mathrm{G}(x)]$ and using Lemma 18.2 again $\mathrm{G}(x) \in [y, \mathrm{G}(y)]$. If $\mathrm{G}(x) \notin [y, b)$, we would contradict Lemma 18.1, so $\mathrm{G}(x) \in [b, \mathrm{G}(y)]$. ◀

An *optimal solution* refers to a set of points $y_0, \ldots, y_{k^*-1}$ on the boundary such that they form a partition of ∂P into k^* visible intervals, i.e. $[y_i, y_{i+1}] \in \mathcal{V}$ and $\bigcup_{i=0}^{k^*} [y_i, y_{i+1}] = \partial P$ and k^* is minimal with this property. Next, we show that any *greedy step*, $(x, \mathrm{G}(x)]$, always contains at least one point from every optimal solution (similar to Lemma 2.3 in [20]).

▶ **Corollary 19** (Greedy steps contain optimal endpoints)**.** *Let* $y_0, y_1, \ldots, y_{k^*-1}$ *be an optimal solution and* $x \in \partial P$*. Then* $(x, G(x)]$ *contains at least one* y_i.

Proof. Let $x \in [y_i, y_{i+1}]$, then $y_{i+1} \in [x, G(x)]$ by Lemma 18.2. If $y_{i+1} \neq x$ we are done (Figure 14 left). If $y_{i+1} = x$ we must have $[x, y_{i+2}] \in \mathcal{V}$ which by Lemma 17.3 implies that $[x, y_{i+2}] \subseteq [x, G(x)]$, and since $y_{i+1} \neq y_{i+2}$, we have $y_{i+2} \in (x, G(x)]$ (Figure 14 right). ◀

Iteratively computing greedy steps is exactly Algorithm 1, leading to the following definition.

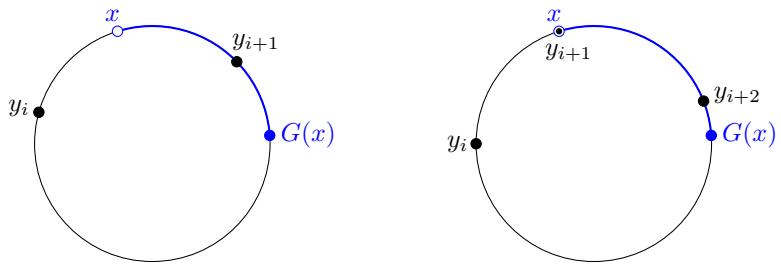


Figure 14 All greedy steps contain a point from every optimal solution.

▶ **Definition 20** (Greedy Sequence). *Given a point x on ∂P, we denote the sequence $(x_i)_{i=0}^{\infty}$ with $x_0 = x$ and $x_{i+1} = G(x_i)$ as the* greedy sequence *starting at x. This is a sequence of endpoints of intervals. If none of the endpoints belong to an optimal solution then we call it a* non-optimal *greedy sequence. Otherwise, it is an* optimal *greedy sequence.*

Note that the greedy sequence $(x_i)_{i=0}^{\infty}$ is infinite, whereas Algorithm 1 only computes a prefix of this sequence. The goal of the rest of the paper is to show that for $T = \Omega(poly(n))$, then the sequence $(x_i)_{i=0}^{T}$ is optimal. Denote by *revolution* a single traversal around ∂P in $k+1$ steps, where k is the minimal number of greedy steps such that $\mathrm{G}(x_k) = x_{k+1} \in [x_0, x_1)$.

Next, we show that the solution found after the first revolution is at most one interval longer than an optimal solution of size k^* (similar to Theorem 2.4 in [20]).

▶ **Theorem 21** (One revolution is at most one-off optimal). *Running Algorithm 1 from any point $x \in \partial P$ and stopping once $x_{k+1} \in [x_0, x_1]$ guarantees a solution of size $k \leq k^* + 1$.*

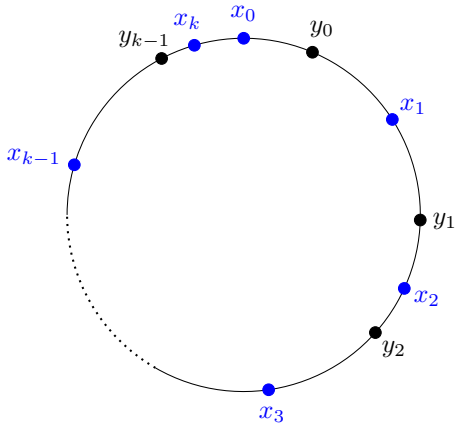


Figure 15 Each greedy interval $(x_i, x_{i+1}]$ for $i = 0, 1, 2, ..., k-1$ is disjoint and contains at least one point from any optimal solution by Corollary 19. This is not guaranteed for the last interval, i.e. the interval $(x_k, x_0]$ at the start of the next revolution which may not be a greedy interval.

Proof. Let $(x_i)_{i=0}^{\infty}$ be the greedy sequence starting at x. Assume that $k > 1$ is minimal such that $x_{k+1} \in [x_0, x_1]$. Thus this candidate solution uses $k+1$ guards.

Let $y_0, \ldots, y_{k^*-1}$ be an optimal solution. From Corollary 19 we know that for each $i = 0, 1, \ldots, k^* - 1$: $y_i \in (x_i, x_{i+1}]$, up to renumbering of the indices; see Figure 15. All these intervals are disjoint; hence an optimal solution contains at least $k-1$ endpoints, i.e. at least $k-1$ guards. Thus the greedy sequence yields a solution of size at most $k^* + 1$ ◀

Theorem 21 is very powerful since we now only need to distinguish between the case where Algorithm 1 is optimal and the case where it uses one extra guard. We now consider how the greedy sequence behaves in different settings, first if x_0 is an endpoint of an optimal solution:

▶ **Lemma 22.** *Let $x \in \partial P$ be a point in some optimal solution. Then Algorithm 1 returns an optimal solution in one revolution starting from x.*

Proof. Let $y_0, y_1, \ldots, y_{k^*-1}$ be an optimal solution and $x = y_0$. Consider the first k^* terms of the greedy sequence starting at x: $x_0, x_1, \ldots, x_{k^*-1}$, see Figure 16.

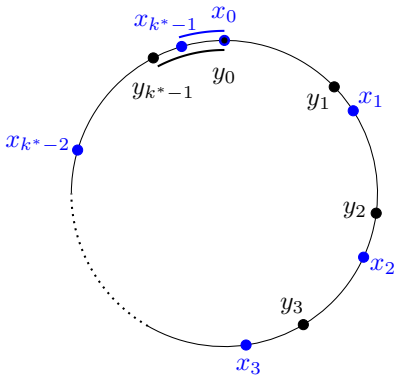


Figure 16 An optimal solution $y_0, y_1, y_2, \ldots, y_{k^*-1}$ and the greedy sequence $x_0, x_1, x_2, \ldots, x_{k^*-1}$ starting from $x_0 = y_0$, that is also optimal since $[x_{k^*-1}, x_0] \subseteq [y_{k^*-1}, y_0]$ and $[y_{k^*-1}, y_0] \in \mathcal{V}$.

Since we know the length is optimal and the intervals $[x_0, x_1], \ldots, [x_{k^*-2}, x_{k^*-1}]$ are visible, it is sufficient to show that $[x_{k^*-1}, x_0]$ is visible.

We have $x_0 = y_0$ by assumption and using Corollary 19 we know that each of the intervals $(x_{i-1}, x_i]$ contains a y_j for each $i = 1, 2, \ldots, k^* - 2$. All these intervals are disjoint, and hence $y_i \in (x_{i-1}, x_i]$ for $i = 1, 2, \ldots, k^* - 2$. Since $y_0, \ldots, y_{k^*-1}$ is an optimal solution we have $[y_{k^*-1}, y_0] \in \mathcal{V}$. But since $y_{k^*-1} \in (x_{k^*-2}, x_{k^*-1}]$ we must have $x_{k^*-1} \in [y_{k^*-1}, y_0]$ and $[x_{k^*-1}, x_0] \subseteq [y_{k^*-1}, y_0] \in \mathcal{V}$, implying that $[x_{k^*-1}, x_0] \in \mathcal{V}$ and $x_0, \ldots, x_{k^*-1}$ is an optimal solution. ◀

It follows immediately from Lemma 22 that once a greedy sequence reaches a point from an optimal solution, it will remain optimal from that point on.

▶ **Corollary 23** (Once optimal, always optimal)**.** *Let $x \in \partial P$ and consider the greedy sequence starting at x. Assuming that there is some i such that $x_i, x_{i+1}, \ldots, x_{i+k^*-1}$ is an optimal solution then $x_{i+j}, \ldots, x_{i+k^*-1+j}$ will be an optimal solution for all $j \geq 0$.*

Proof. The proof follows by induction in j. For $j = 0$ the claim is trivial.

For the induction step, we assume $x_{i+j}, \ldots, x_{i+k^*-1+j}$ is optimal. Thus x_{i+j+1} is an endpoint for an optimal solution, thus by Lemma 22 the greedy solution starting at x_{i+j+1} is optimal, which shows that the solution $x_{i+j+1}, \ldots, x_{i+k^*-1+j+1}$ is optimal. ◀

We are now ready to determine when a solution is optimal. The first condition we establish is if the greedy sequence repeats in the first revolution:

▶ **Lemma 24** (Exact cover is optimal)**.** *Let $x \in \partial P$, then if $x_i, x_{i+1}, \ldots, x_{i+k}, x_i$ appear in the greedy sequence during a single revolution, then $x_i, \ldots, x_{i+k}$ is an optimal solution.*

Proof. Let $y_0, y_1, \ldots, y_{k^*-1}$ be an optimal solution. Each of the intervals $(x_{i+j}, x_{i+j+1}]$ (for $j = 0, \ldots, k-1$) and $(x_{i+k}, x_i]$ must contain an endpoint from an optimal solution by Corollary 19. Thus, $k^* = k+1$ and $x_i, x_{i+1}, \ldots, x_{i+k}$ is optimal. ◀

▶ Remark 25. The first $(x_0, x_1]$ and last step $(x_k, x_{k+1}]$ in a revolution of ∂P can overlap, hence an optimal point y_0 can satisfy the condition of Corollary 19 for both resulting in an $k^* + 1$ approximation. This is impossible when the greedy sequence repeats, see Figure 17.

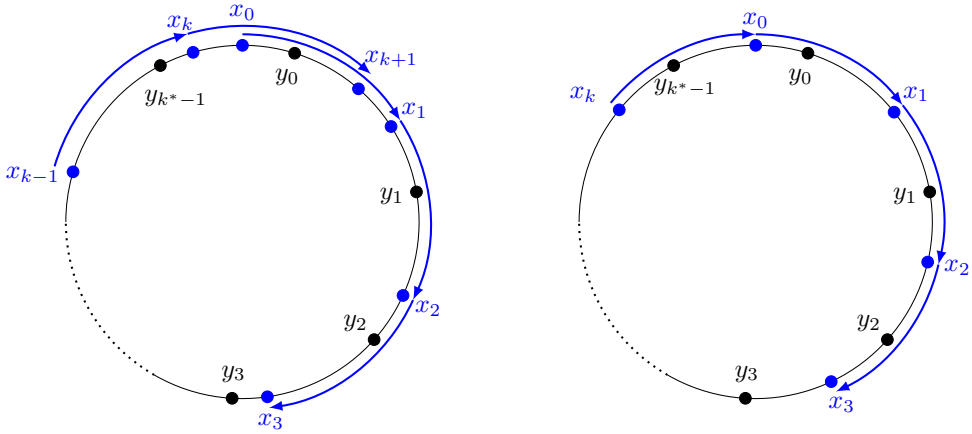


Figure 17 Left, a greedy sequence without repetitions where y_0 appears in both $(x_0, x_1]$ and $(x_{k-1}, x_k]$ which explains how a greedy revolution might not be optimal. Right, a greedy sequence that repeats after each revolution. Greedy intervals are visualized as directed circle arches.

We now know that if the greedy sequence repeats after a single revolution of the polygon, we have an optimal solution. However, to determine that a solution is optimal, we will need weaker conditions than repeating after just one revolution. To do this, we investigate how a non-optimal greedy sequence traverses ∂P, in relation to an optimal solution, see Figure 18.

▶ **Proposition 26** (Behavior of non-optimal greedy sequences). *Let $x \in \partial P$ and consider the non-optimal greedy sequence $(x_i)_{i=0}^{k \cdot N}$ starting at x. Let $y_0, y_1, \ldots, y_{k^*-1}$ be an optimal solution with $y_0 \in (x_0, x_1]$. Then $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m})$ for all $m \in \{1, \ldots, k\}$ and $i \in \{1, 2, \ldots, N-1\}$.*

Proof. Like in the proof of Theorem 21, we distribute the points of an optimal solution y_i in the first k greedy steps. We can do this since the greedy sequence is not optimal implying that the first k greedy steps do not complete the first revolution. Theorem 21 implies that $x_{k+1} \in [x_0, x_1]$. We have $y_i \in (x_i, x_{i+1}]$ for $i = 0, \ldots, k-1$. Now $x_{k+1} = \mathrm{G}(x_k)$ is the first element of the greedy sequence to lie in $[x_0, x_1]$. More specifically, we must have $x_{k+1} \in [y_0, x_1]$, because if not then $x_{k+1} \in [x_0, y_0)$ and $[y_{k-1}, y_0] \not\subseteq (x_{k-1}, x_k]$ and since $[y_{k-1}, y_0]$ is visible, we contradict Lemma 17.3.

We now show $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m}]$ for $i \leq N$ and $m \in \{1, \ldots, k\}$ by induction in $ik + m$. The induction start is exactly $x_{k+1} \in [y_0, x_1]$. For the induction step, assume that $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m}]$. If $m = k$ we will have a carry when we take a step (i.e. increment i by 1 and set $m = 1$), otherwise, we will simply add one to m. We assume we are in the second case to ease notation, but the same argument holds in the carry case.

It is clear, that $x_{(i-1)k+m} \in [y_{m-1}, y_m]$, combining this with $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m}]$ and Lemma 18.3 we get $\mathrm{G}(x_{ik+m}) \in [y_m, \mathrm{G}(x_{(i-1)k+m})]$, i.e. $x_{ik+m+1} \in [y_m, x_{(i-1)k+m+1}]$, see Figure 18.

Finally, since we assumed that none of the candidate solutions are optimal within kN greedy steps, the sequence will not repeat in one revolution as this would contradict Lemma 24, thus $x_{ki+m} \in [y_{m-1}, x_{k(i-1)+m})$ for all relevant $i < N$ and $m \in \{1, 2, \ldots, k\}$. ◀

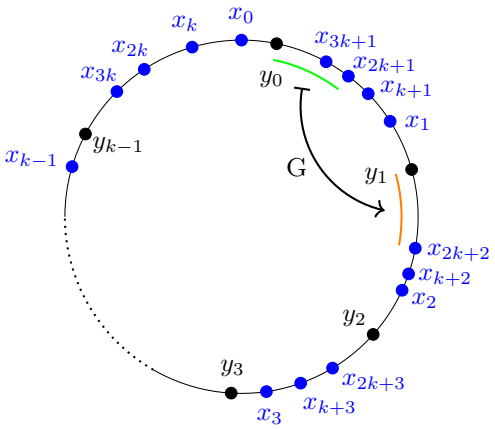


Figure 18 For a non-optimal greedy sequence G maps points from $[y_0, x_{2k+1}]$ (green) to $[y_1, x_{2k+2}]$ (orange), and in the next revolution from $[y_0, x_{3k+1}]$ to $[y_1, x_{3k+2}]$.

▶ Remark 27. It is clear that $x_{ik+m} \in [y_{m-1}, x_{(i-1)k+m})$ implies $x_{ik+m} \in [x_{m-1}, x_{(i-1)k+m})$. This is relevant in practice, as we do not know the location of the y_i's. Thus, the result of Proposition 26 can be restated as $x_{ik+m} \in [x_{m-1}, x_{(i-1)k+m})$ (these intervals are similar to the zones defined in [20]).

Proposition 26 shows that there is an important structure in the greedy sequence when viewed locally, which we capture in the following definitions:

▶ **Definition 28** (Local Greedy Sequence). *Let $x \in \partial P$, consider the greedy sequence $(x_i)_{i=0}^{\infty}$ starting at x. Then $(x_i^m)_{i=0}^{\infty} = (x_{ik+m})_{i=0}^{\infty}$ for $m = 1, \ldots, k$ are* local greedy sequences.

For non-optimal greedy sequences, its local greedy sequences move counterclockwise around ∂P by Lemma 26. We make this precise in the following definition.

▶ **Definition 29** (Fingerprint). *Let $(x_i^m)_{i=0}^{\infty}$ be the m'th local greedy sequence. For $i \geq 1$ we say that x_i^m has a* negative fingerprint *if $x_i^m \in [x_{m-1}, x_{i-1}^m)$. Otherwise, we say that x_i^m has a* positive fingerprint.

With the notion of fingerprints, we introduce *combinatorial optimality conditions*, which if satisfied by a greedy sequence guarantees that it is optimal, that is, it contains an optimal solution. The contrapositive of Proposition 26 gives the first such condition: A local greedy sequence element with a positive fingerprint is optimal.

▶ **Corollary 30** (Positive fingerprint implies optimality). *Let $x \in \partial P$. Assume that x_i^m has a positive fingerprint. Then $x_i^m, x_i^{m+1}, x_i^{m+2}, \ldots, x_{i+1}^{m-1}$ is an optimal solution.*

Proof. Proposition 26 states that if the sequence $x_0, \ldots, x_{ik+m}$ has no subsequence that constitutes an optimal solution, then all individual points have negative fingerprints. The contrapositive of this statement is that if a point x_j in the sequence has a positive fingerprint, then it is a point in an optimal solution, which in turn, by Lemma 22 implies that $x_i^m, \ldots, x_{i+1}^{m-1}$ is an optimal solution. ◀

The second combinatorial optimality condition states that if the local greedy sequence has moved out of the interval of the first revolution, then it is optimal.

▶ **Corollary 31** (Escaping an interval implies optimality). *Let $x \in \partial P$. If an element x_i^m of a local greedy sequence is not in $[x_{m-1}, x_m)$, then x_i^m is the starting point of an optimal solution.*

Proof. As $[x_{m-1}, x_m) \subseteq [x_{m-1}, x_{ik+m})$ for all i, this is a direct consequence of Remark 27. ◀

The third combinatorial optimality condition is perhaps the strongest; it states that if the greedy sequence has a periodic subsequence of any length, i.e., contains a repetition, then it is optimal.

▶ **Corollary 32** (Periodicity implies optimality). *Let $x \in \partial P$ and $(x_i)_{i=0}^{\infty}$ be the greedy sequence starting at x. Assume there exists two indices $i < j$, such that $x_i = x_j$. Then $x_j, x_{j+1}, \ldots, x_{j+k^*-1}$ is an optimal solution.*

Proof. Assume for contradiction that none of $x_0, \ldots, x_j$ are optimal. Let for each $\ell > 0$

$$V_\ell = \bigcup_{m=1}^{k} [y_{m-1}, x_{\ell-1}^m).$$

After ℓk greedy steps, all future points in the greedy sequence will be contained in V_ℓ by Proposition 26, and since $V_\ell \cap \{x_t \mid t = 0, \ldots, \ell k\} = \emptyset$, a point in the greedy sequence cannot be repeated. This is a contradiction, so one of the x_t must be an endpoint of an optimal solution. So by Corollary 23 $x_j, x_{j+1}, \ldots, x_{j+k^*-1}$ is an optimal solution. ◀

We have established conditions in Corollary 30, 31 and 32 that characterize when we can guarantee that the greedy sequence is optimal. However, using only combinatorial insights, we still cannot distinguish between optimal and non-optimal greedy sequences (see Example 58 in Appendix C.1) nor guarantee that running Algorithm 1 long enough yields an optimal solution (See Figure 19). Thus, we return to the geometric setting with Corollary 30, 31 and 32 as optimality conditions.

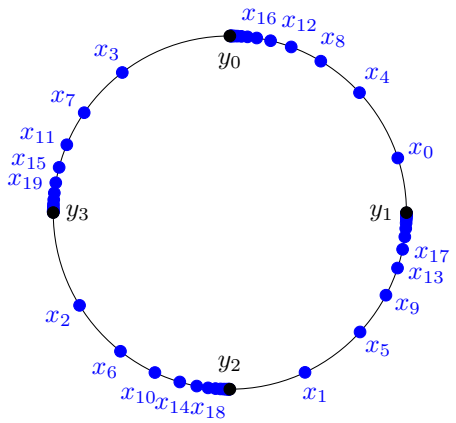


Figure 19 Visualization of a greedy sequence, that never reaches a combinatorial optimality condition since the local greedy sequences converge to their respective y's but never reach them.

5 Algorithm 1, a geometric viewpoint

In this section, we explore the behavior of Algorithm 1 in the geometric setting. We observe the behavior of the (local) greedy sequences by adapting the combinatorial optimality conditions Corollaries 30, 31 and 32 to include geometric observations and finally show that the sequence contains an optimal solution within a polynomial number of revolutions.

First, Corollary 30 and 31 can be related in the geometric to when local greedy sequences move onto new edges:

▶ **Definition 33** (Edge jumps). *Let $(x_i^m)_{i=0}^\infty$ be a local greedy sequence. We say that $(x_i^m)_{i=0}^\infty$ is an* edge jump *at i if x_i^m is a vertex of P or x_i^m and x_{i+1}^m are in different edges of P and neither is a vertex of P.*

▶ **Lemma 34** (Maximal edge jumping). *Let $(x_i^1)_{i=0}^\infty, \dots, (x_i^k)_{i=0}^\infty$ be the local greedy sequences. If there are at least $n+1$ edge jumps across all the local greedy sequences then some local greedy sequence $(x_i^m)_{i=0}^\infty$ will lie outside $[x_{m-1}, x_m]$ after these edge jumps.*

Proof. Let n_m be the number of vertices of P in (x_{m-1}, x_m) and n_{k+1} the number of vertices in (x_k, x_0). Then $\sum_{m=1}^k n_m \leq \sum_{m=1}^{k+1} n_m \leq n$. If the local greedy sequence $(x_i^m)_{i=0}^\infty$ jumps $n_m + 1$ times it will at some point have passed x_{m-1} and thus one of the points must lie in $[x_{m-2}, x_{m-1})$. The lemma follows by the pigeonhole principle. ◀

This leads to a new progress condition based on edge jumps:

▶ **Corollary 35** (Many edge jumps implies optimal). *Let $(x_i)_{i=0}^N$ be a greedy sequence whose local greedy subsequences $n+1$ edge jumps. Then the revolution after x_N will be optimal.*

Proof. The proof is immediate from Lemma 34 and Corollary 31. ◀

▶ Remark 36 (Geometric progress conditions). We establish three progress conditions on the basis of our combinatoric optimality conditions and extend them with geometric results. If one of these are satisfied, we discover an optimal solution or make progress towards one.

1. If we see a positive finger print, the greedy sequence is optimal (Corollary 30).
2. If we see a repetition in the greedy sequence, the sequence is optimal (Corollary 32).
3. If we see $n+1$ edge jumps, the greedy sequence is optimal (Corollary 35).

The goal of the remainder of this section is to prove that we eventually make progress:

▶ **Theorem 37** (Algorithm 1 will reach a progress condition). *Running* GreedyInterval *for $\mathcal{O}(kn^2)$ revolutions guarantees the occurrence of a progress condition.*

We will prove Theorem 37 by considering the local behavior of GreedyInterval when applied to one local greedy sequence. We will need terminology to describe this; see Figure 20.

We let $(y_i)_{i=0}^\infty$ and $(z_i)_{i=0}^\infty$ be local greedy sequences with $z_i = \mathrm{G}(y_i)$. We let g_i be the guard found by GreedyInterval, that sees $[y_i, z_i]$. If $(y_i)_{i=0}^\infty$ or $(z_i)_{i=0}^\infty$ jumps edges, we will have a progress condition, so we assume that this does not happen. Let $(y_i)_{i=0}^\infty$ be contained on ℓ_y and $(z_i)_{i=0}^\infty$ be contained on ℓ_z. We also know, that $(y_i)_{i=0}^\infty$ will move up along ℓ_y and $(z_i)_{i=0}^\infty$ will move down along ℓ_z, as we otherwise would get a positive finger print. Of all the z_i it is only possible for z_0 to be a vertex of P. This is important, as we now know that all other z_i will be defined from a *blockage*, i.e. g_i can see no further than z_i, because there is some vertex of P in the way (see Figure 21).

So we have at least one blockage between z_i and g_i, the closest to z_i we denote $C_{z,i}$. If y_{i+1} could see g_i, then g_i could see $[y_{i+1}, z_i]$, thus we would repeat in the greedy sequence

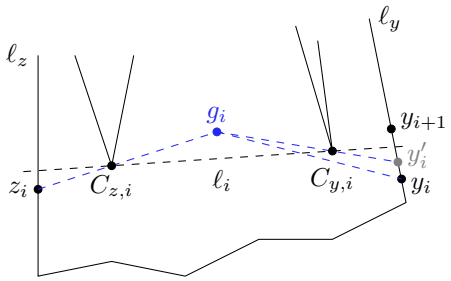


Figure 20 A greedy step from y_i on edge ℓ_y to $z_i := \mathrm{G}(y_i)$ on edge ℓ_z guarded by g_i, however g_i can see from y'_i. The pivot point $C_{z,i}$ blocks g_i from seeing further than z_i and the other pivot $C_{y,i}$ blocks g_i in the other direction, so g_i, $C_{y,i}$ and y'_i lie on a line. The relative position of the line $\ell_i := L(C_{z,i}, C_{y,i})$ between the pivot points to the guard g_i will be vital for our analysis.

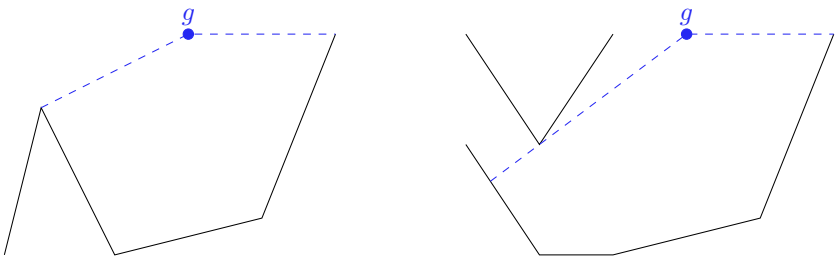


Figure 21 Types of end points with (left) a horizon and (right) a blockage end point

(as $z_i = z_{i+1}$). Assuming this does not happen, there must be something blocking g_i from y_{i+1}. We let y'_i be the point on ℓ_y farthest up, which is still visible from g_i. As it is farther up, there must be some blockage between y'_i and g_i, the closest to y'_i we denote $C_{z,i}$.

We will refer to the points $C_{y,i}$ and $C_{z,i}$ as *pivot points* and $\ell_i := L(C_{y,i}, C_{z,i})$ the *pivot line* for a given i. It is clear that $C_{y,i} \neq C_{z,i}$ so the pivot line is well defined. The pivot points are all vertices of the polygon, thus there are at most $\mathcal{O}(n^2)$ possible pivot lines. In the following, we will look only at one pivot line at a time, and then run enough revolutions to see the same pivot line multiple times by the pigeonhole principle.

Finally, we track where we can place guards. For this, we define feasible regions:

▶ **Definition 38** (Feasible region). *Let $y, z \in \partial P$ and $[y, z]$ the interval of P from y to z. Then the* feasible region $F([y, z])$ *is the subset of P of all possible guard placements that see $[y, z]$. Specifically, $F([y, z]) \neq \emptyset$ if and only if $[y, z] \in \mathcal{V}$.*

In the following, for a greedy step $\mathrm{G}(y_i) = z_i$ where y_i and z_i are on edges ℓ_y and ℓ_z, we define $F := F([y, z])$, where y and z are the endpoints of edges ℓ_y and ℓ_z contained in $[y_i, z_i]$.

Strategy

We briefly outline our approach, which is based on the relationship between the feasible region and the pivot line in a greedy step, $\mathrm{G}(y_i) = z_i$. In Section 5.2, we show that the case where F is entirely above ℓ_i will lead to a progress condition. In Section 5.3, we show that if the feasible region F contains points below ℓ_i, then within the next revolution a different pivot line $\widehat{\ell_i}$ is visited, with corresponding feasible region $\widehat{F}$ entirely above $\widehat{\ell_i}$.

5.1 Supporting lemmas

Throughout this section, we will need the following two properties:

▶ **Lemma 39.** *For any $a, b \in \partial P$, the feasible region $F([a, b])$ is connected. Furthermore, for any convex subset $C \subset P$, $C \cap F([a, b])$ is convex.*

Proof Sketch. The proof follows by induction in the number of vertices in $[a, b]$. In the base case, the feasible region is a single visibility polygon, hence it is connected. In the induction step, we intersect the previous feasible region with a visibility polygon from a vertex v_{i+1}. If this new feasible region F were to be disconnected, we find two points S, T in different components of F, where $\overline{ST}$ is contained in a triangle completely visible to v_{i+1}, i.e. completely contained inside P. This will contradict Lemma 55 in Appendix B.1 and thus F will be connected. A similar argument shows in convexity claim.

For the detailed proof, see Appendix B.1. ◀

▶ **Lemma 40** (Blockings happen outside visible area)**.** *Let $y \in \partial P$ and $z = G(y)$ and assume that neither y nor z is a vertex of P. Let g be a guard seeing $[y, z]$ and let c be a vertex blocking the view from g to z. Then $c \notin [y, z]$.*

Proof Sketch. One considers what g can see around C. It can be shown, that g cannot see everything in the vicinity of C (Lemma 57 in Appendix B.2). Assume for contradiction that $C \in [y, z]$, then either some part of $[y, z]$ is not visible from g or z can be moved to a visible vertex, yielding a contradiction.

For the detailed proof, see Appendix B.2. ◀

5.2 Feasible region strictly above pivot line

In this section, we analyze configurations in which the feasible region lies strictly above a pivot line. The main result of this section is to show that the guard will have a unique optimal position in F. This will imply a progress condition under the assumption that the same pivot line is chosen twice. In Figure 22 we see that the angles between the pivot points and the points above the pivot line are a good measure of the quality of prospective guard positions. In Figure 23 we use this insight to define the *shadow* of a point, which is used in Lemma 43 to show that the feasible region is contained in the shadow of one of its unique point closest to ℓ_i. Finally, in Proposition 44 we show that whenever the feasible region is above the pivot line, a progress condition will be satisfied.

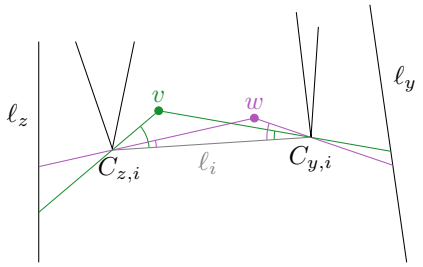


Figure 22 In the setup of Figure 20, a guard above the pivot line ℓ_i placed in point v can see more of the ℓ_y edge than if it was placed in w, and vice-versa, analogously to Figure 2.

Fixing v, it is clear that the area where a guard will have a smaller angle than v with respect to ℓ_i is below $L(v, C_{z,i})$, and a larger angle above $L(v, C_{z,i})$ see Figure 23.

The intersection of the half-planes above $L(a, C_{z,i})$ and $L(v, C_{y,i})$ is a quarter plane where every guard placement is worse than v, both with respect to ℓ_y and ℓ_z. For a point $v \in F$ we denote this quarter plane $S(v)$ and call it the *shadow* of v; see Figure 23 (right).

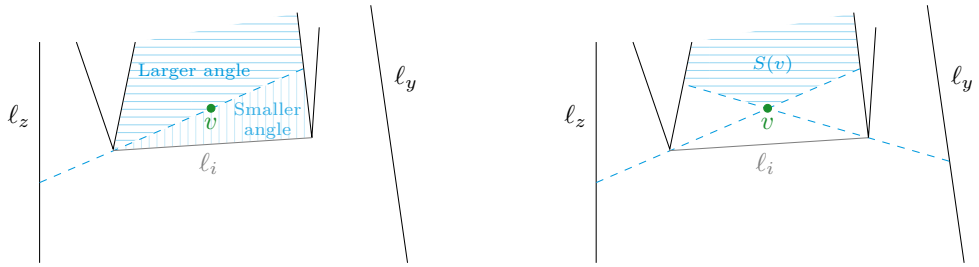


Figure 23 Left, points with smaller angle than v with ℓ_i see more of ℓ_z and are better guards. Right, placing a guard strictly in the shadow $S(v)$ of v leads to a strictly worse guard.

▶ Remark 41 (z above ℓ_i). Let z be the vertex of ℓ_z in $[y_i, z_i]$ and a is the vertex of F closest to ℓ_i. If z lies above $L(a, C_{z,i})$, as illustrated in Figure 24, we see some weird behavior:

For $C_{z,i}$ to be the pivot, z_i has to lie below ℓ_i, thus a cannot see any part of ℓ_z except for z. Thus it makes sense to force the points in F to see a small part of ℓ_y and ℓ_z. This can be done by setting $F = F([y - \varepsilon, z + \varepsilon])$, where $y - \varepsilon$ and $z + \varepsilon$ refers to points on ℓ_y respectively ℓ_z very close to y respectively z.

Making this change will not impact the other proofs other than fixing some special cases in this subsection. By doing this, z cannot lie above $L(a, C_{z,i})$, which is important for future proofs. The analogous behavior holds true if y lies above $L(C_{y,i}, a)$.

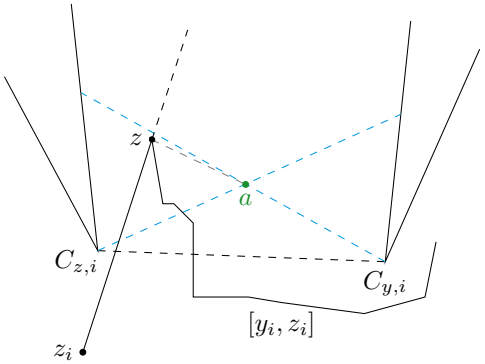


Figure 24 a can see nothing on ℓ_i but z. Thus picking a as a guard would lead to edge jumps immediately

▶ **Lemma 42** (Successor and predecessor edges to $a \in F$ lie in $S(a)$)**.** *Assume F lies strictly above ℓ_i and let a be a vertex of F closest to ℓ_i. Let e and f denote the edges of F connected to a. Then $e, f \subseteq S(a)$.*

Proof. We will start by determining what characterizes edges in F. Since F is the intersection of the visibility polygons of vertices of ∂P, an edge of F has to be an part of an edge of the visibility polygon to a vertex v.

The edges of the visibility polygon from v come in two ways: Either they come from shooting rays from v through other vertices of P or as edges of P.

Assume for contradiction that e or f is not contained in $S(a)$. As the proof is symmetric, we assume $e \not\subseteq S(a)$, e's other endpoint is b and b is closer to $C_{y,i}$ than $C_{z,i}$ (again by symmetry). Furthermore let z be the last vertex of $[y_i, z_i]$, thus z can see F and hence also points a and b. Now e must either be a segment contained in a ray from a vertex $v \in [y_i, z_i]$ or e is a part of ∂P.

▷ Case 1. If $e \subseteq \overrightarrow{v\,a}$ we must have $v \in L(b, a)$. For z to see a, z must lie above $L(b, a)$ as $[v, z]$ would block it otherwise. However now we would need z above $L(a, C_{z,i})$, which we disallow by Remark 41. Thus we arrive at a contradiction.

▷ Case 2. If e is a part of ∂P, we can assume F lies above $L(b, a)$, since a is a lowest point of F; see Figure 25 (right). However, now z cannot see b as the space directly below $\overline{a\,b}$ is not inside P and Remark 41 forcing z below $L(b, a)$, leading to a contradiction. ◀

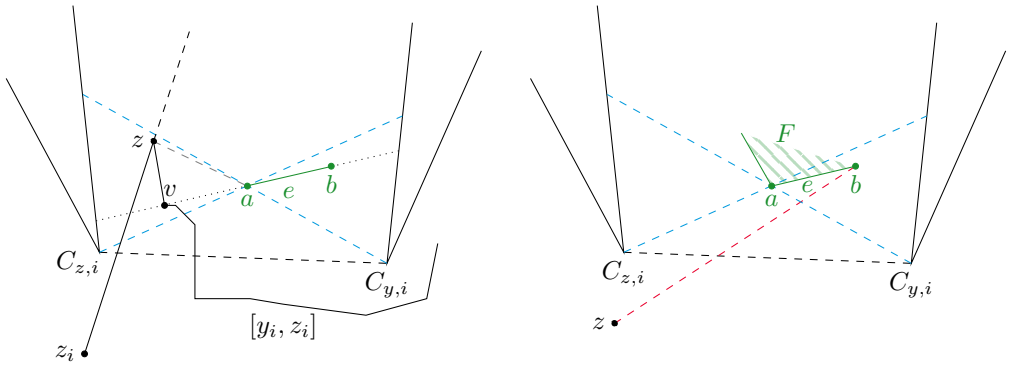


Figure 25 Left, if z sees a, z is above $L(b, a)$ and z_i is below ℓ_i for $C_{z,i}$ to be a pivot. Right, if nothing blocks visibility between z and b then a is not closest to $\ell_i = \overline{C_{y,i}\,C_{z,i}}$.

▶ **Lemma 43** (The feasible region is contained in the shadow). *Assume that F lies above ℓ_i and let a be a vertex of F closest to ℓ_i. Then $F \subseteq S(a)$.*

Proof. Assume for contradiction there are points in F outside of $S(a)$. Let b be a point in F such that $\angle baC_{y,i}$ or $\angle C_{z,i}ab$ is minimal (depending on whether b is closer to $C_{y,i}$ or $C_{z,i}$). Assume w.l.o.g. b is closer to $C_{y,i}$ than $C_{z,i}$. Because of the minimality of $\angle baC_{y,i}$, we now know all of F is above $L(b, a)$.

By Lemma 42 we know b is not the successor or predecessor to a, thus not all of $\overline{a\,b}$ will be in F. So for at least one vertex v, a and b will be visible, but not all of $\overline{a\,b}$. Let d be a point on $\overline{a\,b}$ not visible to v.

It must now hold that F cannot intersect the ray $\overrightarrow{v\,d}$. Furthermore, F cannot cross $L(b, a)$ from the minimality of the angle (see Figure 26). Thus a and b must be in disconnected components of F. By Lemma 39 F must be connected, leading to a contradiction. ◀

We can now show that when F lies strictly above ℓ_i, we get a progress condition:

▶ **Proposition 44** (Feasible region over repeated pivot line implies a progress condition). *If $\ell_i = \ell_j$ for some $i < j$ and F lies above ℓ_i, then we get a progress condition.*

Proof. Let v be a vertex of F closest to ℓ_i. By Lemma 43 we know $F \subseteq S(v)$.

Consider y_i. If it can see any part of F, it can also see v, since $F \subseteq S(v)$, and any point in F can see no farther than v on ℓ_z, again because $F \subseteq S(a)$. Thus, z_i is the farthest point

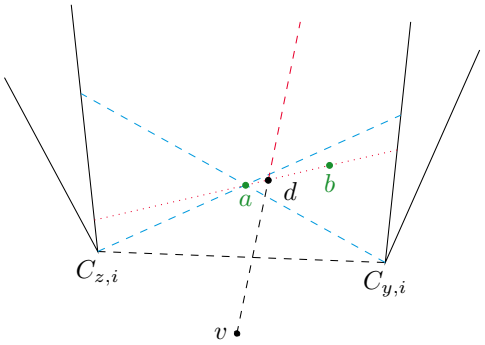


Figure 26 Ray $\overrightarrow{v\,d}$ and $L(b,a)$ divides the feasible region F into disconnected components

on ℓ_z visible from v. If y_i could not see any point in F, then z_i could not lie on ℓ_z which is an edge jump, hence a progress condition.

Now consider y_j. We get the exact same considerations as above, so z_j is the farthest point on ℓ_z visible from v and $z_i = z_j$, a repetition, or $z_j \notin \ell_z$, and edge jump, both are progress conditions, as wanted. ◀

5.3 Feasible region below pivot line

In this section, we analyze what happens when F is not strictly above ℓ_i. Contrary to what we have seen in Section 5.2, when g_i can lie below ℓ_i, it does not hold that there is a unique optimal guard placement. Thus we will show that when F is on or below ℓ_i, we will be able to find some other guard, which is found by GreedyInterval, that lies strictly above its pivot line, thus forcing a unique optimal guard placement at some other point in the polygon. We denote the polygon constructed by the edges of $[y_i', z_i]$, $\overline{z_i\,g_i}$ and $\overline{g_i\,y_i'}$ by Q_i, see Figure 27. Recall that y_i' is defined considering the last point on ℓ_y visible to g_i.

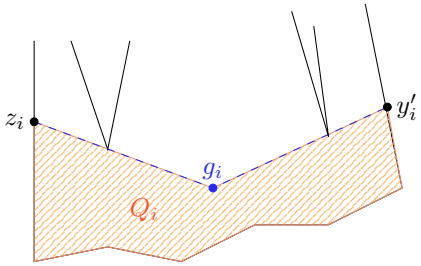


Figure 27 The polygon Q_i defined by $[y_i', z_i]$, $\overline{z_i\,g_i}$ and $\overline{g_i\,y_i'}$.

▶ **Lemma 45.** *$[z_i, y_i']$ does not cross the boundary of Q_i.*

Proof. The guard g_i must be able to see y_i' and z_i, so $\overline{g_i\,y_i'}$ and $\overline{g_i\,z_i}$ must not be obstructed. Since all the boundary segments in $[y_i', z_i]$ cannot be obstructed either as P is simple, no part of $[z_i, y_i']$ (i.e. the interval not guarded by g_i) can cross the boundary of Q_i. ◀

If F lies above and below ℓ_i we could potentially have guards below and above. However, the following lemma shows that guards are below the pivot line if possible.

▶ **Lemma 46** (Guards are placed on or below the pivot lines if possible)**.** *Let ℓ_i^- be the closed half-plane below the i'th pivot line. If $F \cap \ell_i^- \neq \emptyset$ then g_i will lie on or below ℓ_i.*

Proof. Assume for contradiction that g_i lies strictly above ℓ_i. It holds that F is connected by Lemma 39. Combining this with the $F \cap \ell_i^- \neq \emptyset$ condition, then there must be a point on $\overline{C_{y,i}\, C_{z,i}} \subset \ell_i$ also contained in F (ℓ_i is the extended line, where $\overline{C_{y,i}\, C_{z,i}}$ is only the line segment). Since $\overline{C_{y,i}\, C_{z,i}} \subseteq Q_i$ we know that $F \cap \ell_i^- \cap Q_i \neq \emptyset$. Let $g_i' \in F \cap \ell_i^- \cap Q_i$.

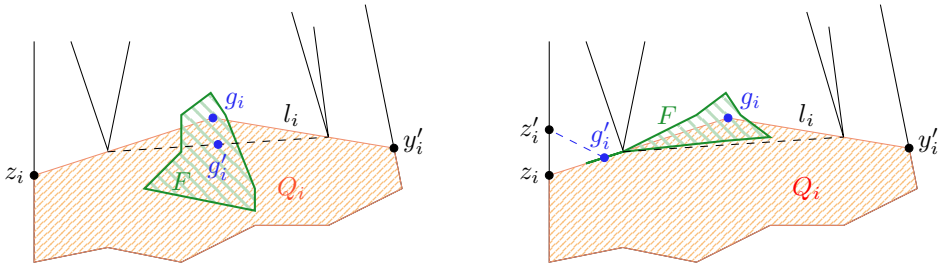


Figure 28 Left, the setup of Lemma 46 with feasible region F, guard g_i that can see all of Q_i along with placement of g_i'. Right, g_i cannot be optimal as g_i' can see further than g_i.

As $g_i' \in F$, g_i' can see every vertex in $[y_i, z_i]$, and since $g_i' \in Q_i$, g_i' can also see y_i and z_i by Lemma 45 and Lemma 40, see Figure 28 left. Thus g_i' must also see all of $[y_i, z_i]$. However, we know from analyzing GreedyInterval (in Remark 6), that multiple optimal guards must be collinear with z_i, so $g_i' \in L(z_i, g_i) = L(z_i, C_{z,i})$ and since $g_i' \in \ell_i^-$, g_i' must lie in $\overline{z_i\, C_{z,i}}$. But now no vertex of P is between g_i' and z_i (since $C_{z,i}$ is the vertex closest to z_i between z_i and g_i), so g_i' must be able to see more than z_i making g_i not optimal, leading to a contradiction. ◀

Now we know that the placement of the guard must be below ℓ_i, if possible. We next prove that the points y_i and z_i will both be above ℓ_i if we see ℓ_i again:

▶ **Lemma 47** (y_j and z_j will lie above ℓ_j)**.** *If $F \cap \ell_i^- \neq \emptyset$ and $\ell_i = \ell_j$ with $i < j$. Then y_j and z_j will both lie above ℓ_i or we reach a progress condition.*

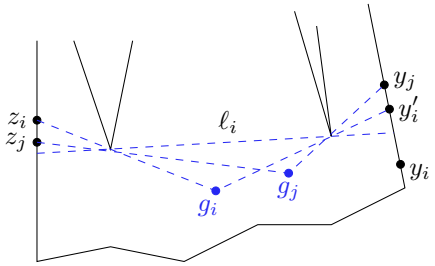


Figure 29 If the pivot line ℓ_i is reused at step $i < j$ then both y_j and z_j are above it.

Proof. We also have that $C_{y,i}$ lies on $\overline{g_i\, y_i'}$, thus y_i' lies above ℓ_i, and since $i < j$, either y_j will also lie above ℓ_i or we get a positive fingerprint.

Analogously, $C_{y,i}$ lies on $\overline{g_i\, y_i'}$, thus y_i' lies above ℓ_i, and since $i < j$ (and we assume negative fingerprint), y_j will also lie above ℓ_i. ◀

We are now ready to give the main result of the subsection, essential in proving Theorem 37.

▶ **Proposition 48** (Guard placed below pivot implies other guard placed strictly above pivot)**.** *Let g_i lies below the pivot line and y_i, z_i lie above the pivot line, then one of the following will occur:*

1. *In the next revolution, we will see a progress condition.*
2. *z_{i+1} will lie below the pivot line ℓ_i.*
3. *There is some guard $\widehat{g}$ found in the next revolution, which is below its pivot line $\widehat{\ell}$.*

Proof. Consider the triangle $\triangle g C_{z,i} C_{y,i}$. Let x be a point on $\partial P \cap \triangle g C_{z,i} C_{y,i}$ which is furthest below ℓ_i (it may be that $x = C_{z,i}$ or $x = C_{y,i}$). During the next $k+1$ greedy steps, we will at some point have a guard, that sees x as part of its interval. If x is the endpoint of such an interval, we have an edge jump. So we assume x is not an endpoint. Next, notice that $x \in [C_{z,i}, C_{y,i}]$, since if $x \in [y_i', z_i]$, g_i would not be able to see y_i' and z_i and if $x \in [z_i, C_{z,i}]$ or $[C_{y,i}, y_i']$ $x \in [z_i, C_{z,i}]$ or $x \in [C_{y,i}, y_i']$, that piece of the boundary would need to enter Q_i contradicting Lemma 45.

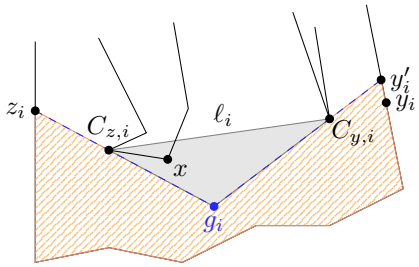


Figure 30 A point $x \in \partial P$ furthest below ℓ_i must lie in the triangle $\triangle g_i C_{z,i} C_{y,i}$.

Let $\widehat{g}$ be the guard that sees x. We introduce the same terminology around $\widehat{g}$ as for g_i: $\widehat{g}$ guards $[\widehat{y}, \widehat{z}]$ and $\widehat{g}$ is blocked by $\widehat{C}_z$ and $\widehat{C}_y$ with $\widehat{y}'$ being the furthest $\widehat{g}$ can see when going backwards. Let $\widehat{\ell} = L(\widehat{C}_y, \widehat{C}_z)$ be the pivot line associated with $\widehat{g}$.

In the following, we will consider where $\widehat{g}, \widehat{C}_z$ and $\widehat{C}_y$ can be placed in relation to each other. We will see that in all these cases one of the three criteria in the proposition will be satisfied.

Let $\widehat{F}$ be the feasible region for the vertex x and the edges of P connected to x. $\widehat{F}$ is obtained by extending the edges connected to x as in Figure 31.

Finally, let z be the vertex on $[y_i, z_i]$ just before z_i.

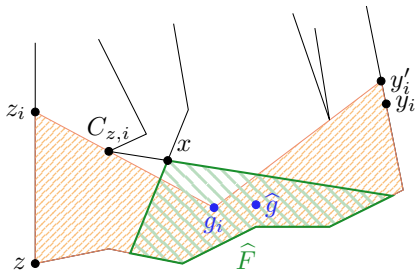


Figure 31 The guard $\widehat{g}$ that guards x must lie in the feasible region $\widehat{F}$ of the two edges that share x as an endpoint.

Since x is chosen lowest in $\triangle C_{z,i} C_{y,i} g_i \cap \partial P$, both edges in ∂P connected to x will point downwards (where ℓ_i is horizontal). Thus $\widehat{F}$, whose boundary is the continuation of these edges, will be contained below ℓ_i.

We now look at the case, where $\widehat{g}$ is not in Q_i:

If $\widehat{g}$ is not in Q_i, it lies above $L(y_i', g_i)$ and $L(g_i, z_i)$, as they are extensions of borders of Q_i. From the considerations above, we know that $\widehat{g}$ must also lie below ℓ_i, thus $\widehat{g} \in \triangle g_i C_{z,i} C_{y,i}$.

Furthermore, we know that x is the lowest point of ∂P inside $\triangle C_{z,i}C_{y,i}g_i$, and $\widehat{g}$ is below x, thus below $\widehat{g}$ there is no part of ∂P inside $\triangle gC_{z,i}C_{y,i}$.

We now assume for contradiction, that on of the pivots for $\widehat{g}$, $\widehat{C}_y$ or $\widehat{C}_z$, lies below $\widehat{g}$. Then $\widehat{y}'$ or $\widehat{z} \in [y'_i, z_i]$, since $\widehat{y}'$, respectively $\widehat{z}$, lie on the ray $\overrightarrow{\widehat{g}\,\widehat{C}_y}$, respectively the ray $\overrightarrow{\widehat{g}\,\widehat{C}_z}$, and these rays will be contained in the region of $\triangle g_i C_{z,i}C_{y,i}$ below $\widehat{g}$ and Q_i (until they hit ∂P), where no part of $[z_i, y'_i]$ can enter by Lemma 45 (see Figure 32 left).

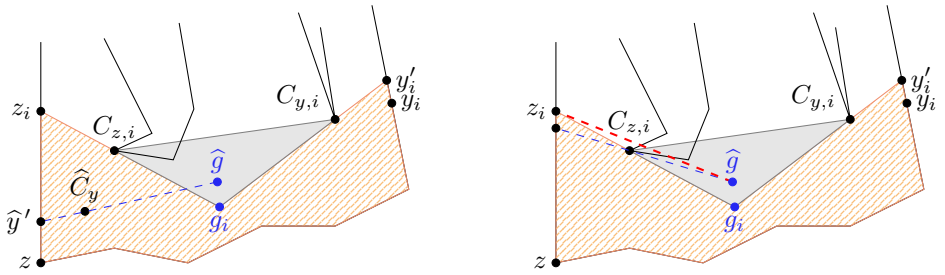


Figure 32 Assuming $\widehat{g}$ is not in Q_i and $\widehat{C}_y$ below $\widehat{g}$, we must have $\widehat{y}$ in $[y'_i, z_i]$ (left). However $C_{y,i}$ blocks the view for $\widehat{g}$ to see z_i (right), so $[\widehat{y}, x]$ cannot both be visible from $\widehat{g}$.

This implies that we would need $\widehat{g}$ to be able to see all of $[z_i, x]$ or $[x, y'_i]$. However this is not possible, as $\widehat{g}$ is strictly above the lines $L(C_{y,i}, g_i)$ and $L(g_i, C_{z,i})$ and thus the pivots $C_{y,i}$ and $C_{z,i}$ will block $\widehat{g}$ from seeing y'_i and z_i. This yields a contradicition.

Now both $\widehat{C}_y$ and $\widehat{C}_z$ are above $\widehat{g}$. From the assumption that $\widehat{g}$ sees x, we know that $x \in [\widehat{y}', \widehat{z}]$, thus $\widehat{C}_y$ will lie to the left of $\overline{\widehat{g}\,x}$ and $\widehat{C}_z$ to the right. So now $\widehat{\ell}$ must lie below $\widehat{g}$ and we have showed what we wanted (see Figure 33).

Now we assume that $\widehat{g}$ lies in Q_i, and again we consider, where $\widehat{C}_y$ and $\widehat{C}_z$ can be located. Since x is in $[\widehat{y}', \widehat{z}]$, we cannot have $\widehat{z} \in (z_i, x)$, as this would contradict the maximality of greedy step from y_i to z_i. Likewise we cannot have $\widehat{y}' \in (x, y_i)$, again due to maximality. The possible placements of $\widehat{z}$ and $\widehat{y}$, along with the possible placements of $\widehat{C}_z$ and $\widehat{C}_y$ are marked on Figure 34.

▷ Case 1. If $\widehat{z} \in (y'_i, y_i]$, then $\widehat{z}$ will be in the same local greedy sequence as y_i, hence $\widehat{z} = y_{i+1}$, but now g_i sees $[y_{i+1}, z_i]$, thus we must repeat endpoints and the greedy sequence repeats, which is one of the desired conditions.

▷ Case 2. If $\widehat{z} \in (y_i, z_i]$, then $\widehat{z}$ must be in the same local greedy sequence as z_i, i.e. $\widehat{z} = z_{i+1}$, implying that $\widehat{y}$ must be in the same local greedy sequence as y_i, so $\widehat{y} = y_{i+1}$, but $\widehat{y}$ is in $[y'_i, C_{z,i}]$, so y_{i+1} either jumps edges, has a positive fingerprint or $z_{i+1} = z_i$, a repetition.

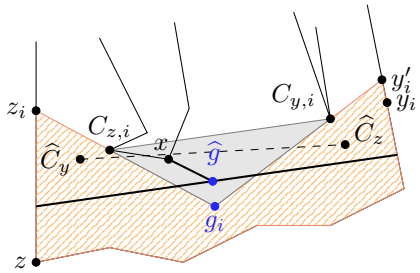


Figure 33 If $\widehat{C}_y$ and $\widehat{C}_z$ are above $\widehat{g}$, and $x \in [\widehat{y}', \widehat{z}]$ then we must have $\widehat{g}$ above $\widehat{\ell}$.

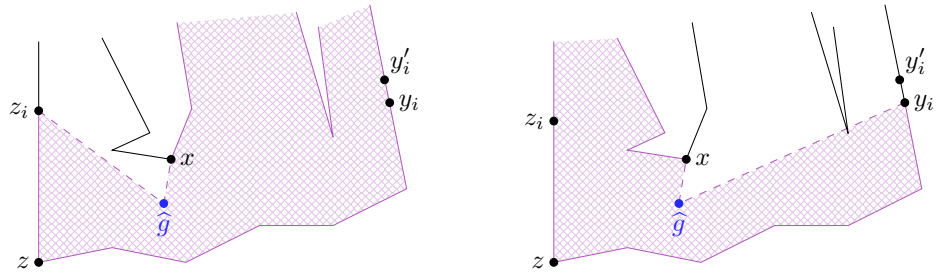


Figure 34 Left, $\widehat{z}$ must lie in $[x, z_i]$ and $\widehat{C}_z$ in the polygon defined by $[x, z_i]$, $\overline{z_i\,\widehat{g}}$ and $\overline{\widehat{g}\,x}$. Right, analogously for $\widehat{y}$ and $\widehat{C}_y$.

▷ Case 3. Now $\widehat{z} \in [x, y_i']$. Consider the subcases where g_i lies above or below $L(y_i', \widehat{g})$: If g_i lies above, $\widehat{C}_z$ must lie above $L(y_i', \widehat{g})$, since we otherwise would have the ray $\overrightarrow{\widehat{g}\,\widehat{C}_z}$ contained in Q_i (until it hits ∂P), since $\widehat{g} \in Q_i$, making $\widehat{z} \in (y_i', z_i]$. If g_i lies below $L(y_i', \widehat{g})$, we must have $\widehat{C}_z$ above $L(y_i', g_i)$ and $L(g_i, \widehat{g})$ for the same reason (see Figure 35).

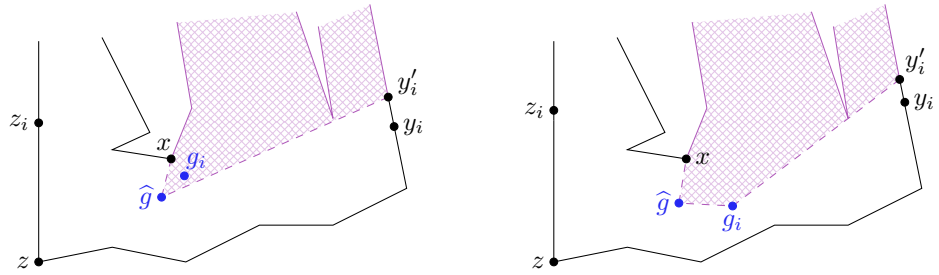


Figure 35 Further restrictions to Figure 34. On the left, the possible placement of $\widehat{z}$ and $\widehat{C}_z$ are marked in red, when g_i lies above $L(y_i', \widehat{g})$. On the right, the possible placements when g_i lies below $L(y_i', \widehat{g})$

We now look at where we can place $\widehat{C}_y$ so that $\widehat{g}$ lies below $\widehat{\ell}$ in both of these cases:

▷ Subcase 3.1 (g_i above $L(y_i', \widehat{g})$). If g_i lies above $L(y_i', \widehat{g})$, we must place $\widehat{C}_y$ below $L(y_i', \widehat{g})$ for $\widehat{\ell}$ to lie above $\widehat{g}$. Combined with the previous restrictions, we see on Figure 36, the possible placements for $\widehat{C}_y$.

Now the ray $\overrightarrow{\widehat{g}\,\widehat{C}_y}$ will be contained in Q_i, so $\widehat{y}' \in [y_i, z_i]$. This means, that $\widehat{y}$ will be in the same local greedy sequence as z_i, thus $\widehat{y} = z_i$. If $\widehat{y}' \in [y_i, z]$, either $z_{i+1} \in [y_i, \widehat{y}']$ and

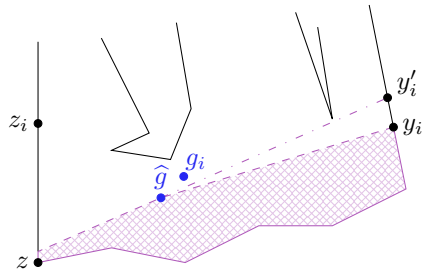


Figure 36 When g_i is over $L(y_i', \widehat{g})$ then $\widehat{y}$ and $\widehat{C}_y$ are restricted by y_i and $L(\widehat{g}, y_i')$

we get an edge jump, or $z_{i+1} \in [\widehat{y}', z_i]$ and we will repeat since $\mathrm{G}(z_i) = \mathrm{G}(z_{i+1})$, both are terminal conditions of the proposition. Thus, it remains to analyze the case $\widehat{y}' \in (z, z_i]$.

We have y_i' above ℓ_i and $\widehat{g}$ below ℓ_i. Thus $\widehat{y}'$ will also lie below ℓ_i. If $z_{i+1} \in [\widehat{y}', z_i]$ we will again get a repetition, so we need $z_{i+1} \in [z, \widehat{y}]$, and z_{i+1} is below ℓ_i, which is the third terminal condition of the proposition.

▷ Subcase 3.2 (g_i below $L(y_i', \widehat{g})$). If g_i lies below $L(y_i', \widehat{g})$ we again consider where to place $\widehat{C}_y$. To the right of $\widehat{g}$, it must be below $L(y_i, \widehat{g})$ by the previous arguments. To the left of $\widehat{g}$ it must be below $L(g_i, \widehat{g})$ for $\widehat{g}$ to lie below $\widehat{\ell}$ (see Figure 37).

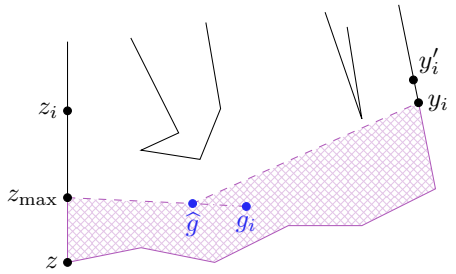


Figure 37 Case where g_i is below $L(y_i', \widehat{g})$: Where to put $\widehat{y}$ and $\widehat{C}_y$

Since $\widehat{g}$ lies in Q_i, we must have that the ray $\overrightarrow{\widehat{g}\, g_i}$ is contained in Q_i. Let the intersection of this ray with l_z be $z_{\max}$. Thus $\widehat{y}' \in [y_i, z_{\max}]$ and we again have $\widehat{y} = z_i$, likewise, if $\widehat{y}' \in [y_i, z]$ we will get an edge jump or repetition as in case 3.1. So we assume $\widehat{y} \in (z, z_i]$.

Now we must have $\widehat{C}_y$ inside the triangle $\triangle \widehat{g} z_{\max} z$, which is contained in Q_i. Thus $\widehat{C}_y \in [y_i, z_i]$. Since $\widehat{C}_y$ lies below $L(g_i, \widehat{g})$ and $\widehat{y}'$ lies on the ray $\widehat{g}\widehat{C}_y$, we must have that $\widehat{y}'$ lies below $L(g_i, \widehat{g})$ and since $\widehat{C}_y \in \overline{\widehat{g}\widehat{y}'}$, $\widehat{C}_y$ must lie above $L(g_i, \widehat{y}')$. It is also clear, that z must lie below $L(g_i, \widehat{y}')$, but $\widehat{C}_y$ and z are connected by edges in ∂P. This now means, that g_i cannot see $\widehat{y}'$, contradicting the fact that g_i is a guard that sees $[y_i, z_i]$ (see Figure 38).

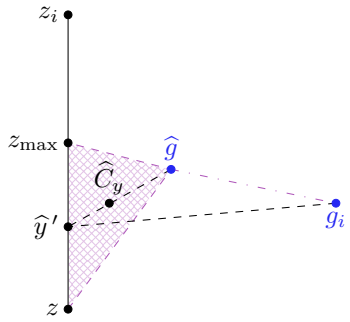


Figure 38 $\widehat{C}_y$ will block g_i's view to $\widehat{y}'$

Considering all the above cases, we show that when $\widehat{g}$ lies below $\widehat{\ell}$ it implies either an edge jump, a repetition, a positive finger print or z_{i+1} being below ℓ_i, as wanted. ◀

We now have all the tools needed to prove Theorem 37, which was the goal of this section.

▶ **Theorem 37** (Algorithm 1 will reach a progress condition). *Running* GreedyInterval *for* $\mathcal{O}(kn^2)$ *revolutions guarantees the occurrence of a progress condition.*

Proof. Assume for contradiction that we see no progress conditions in the first $\mathcal{O}(kn^2)$ revolutions of Algorithm 1. Let $(x_i^m)_{i=0}^{\infty}$ for $m = 1, \ldots, k$ be the local greedy sequences. All these are contained on their respective edges (since we see no edge jumps). Let F^m be the feasible region for the vertices of ∂P in the interval $[x_{m-1}, x_m]$. Furthermore let ℓ_i^m be the pivot line for the pair (x_i^{m-1}, x_i^m). Consider the choice of ℓ_i^1 for $i = 1, \ldots, \mathcal{O}(kn^2)$ when Algorithm 1 makes $\mathcal{O}(kn^2)$ revolutions. We assume for contradiction that we see no progress conditions and consider the following possible cases:

▷ Case 1. If F^1 lies strictly above ℓ_i^1, we cannot repeat ℓ_i^1, as this leads to a progress condition (Proposition 44). Thus this can occur at most $\mathcal{O}(n^2)$ times (i.e. once for each possible pivot line).

▷ Case 2. If F^1 lies at or below ℓ_i^1 we consider whether it is the first time we have seen this pivot line or not:

▷ Subcase 2.1. If it is the first time ℓ_i^1 is a pivot line (i.e. $\ell_i^1 \neq \ell_j^1$ for all $j < i$), we cannot deduce anything. However, this can happen at most $\mathcal{O}(n^2)$ times (i.e. once for each possible pivot line).

▷ Subcase 2.2. If it is not the first time, Lemma 47 states that x_{i-1}^k and x_i^1 lie above ℓ_i^1. Now the conditions of Proposition 48 are satisfied, so we either obtain a progress condition within one revolution, x_{i+1}^1 lies below ℓ_i^1, or some guard g_i^m that lies above ℓ_i^m. We consider the last two cases.

▷ Subsubcase 2.2.1. If x_{i+1}^1 lies below ℓ_i, we cannot use ℓ_i again as this would break Lemma 47. Thus this case can happen at most $\mathcal{O}(n^2)$ times.

We have shown, that at most $\mathcal{O}(n^2)$ of the $\mathcal{O}(kn^2)$ i's can fall into the above cases, hence we still have $\mathcal{O}(kn^2)$ i's left for this final case:

▷ Subsubcase 2.2.2. Finally some other g_i^m lies above its pivot line ℓ_i^m. Since there are $k-1$ other feasible regions, and each of these has $\mathcal{O}(n^2)$ pivot lines, at some point, we will have chosen the same feasible region with the same pivot line pair twice, by the pigeonhole principle. Since the guard is above this pivot line, the entire feasible region will also be above by contraposition of Lemma 46. Now we have seen the same pivot line twice with the same feasible region strictly above it, so Proposition 44 yields a progress condition. ◀

5.4 Proof of main theorem

Finally, we prove Theorem 1:

▶ **Theorem 1.** *The contiguous art gallery problem for a simple polygon with n vertices is solvable in $\mathcal{O}(k^* n^5 \log n)$ arithmetic operations, where k^* is the size of an optimal solution.*

Proof. Running Algorithm 1 for $\mathcal{O}(kn^2) = \mathcal{O}(k^* n^2)$ revolutions guarantees a progress condition i.e. a positive fingerprint, a repetition, or an edge jump by Theorem 37. Thus, repeating $n+1$ times guarantees at least one positive fingerprint, at least one repetition, or $n+1$ edge jumps. A positive fingerprint or repeatetition implies optimality by Corollary 30 and 32. If we have $n+1$ edge jumps, we are optimal by Corollary 35. Now after $\mathcal{O}(k^* n^3)$ revolutions Algorithm 1 will find an optimal endpoint, and by Corollary 23 output an optimal solution. The algorithm makes a revolution in $\mathcal{O}(n^2 \log n)$ arithmetic operations by Corollary 13, the total number of arithmetic operations is $\mathcal{O}(k^* n^5 \log n)$. ◀

6 Bit Complexity of Algorithm 1

In this section we show how to bound the bit complexity of the points calculated by the GreedyInterval algorithm, to then bound the overall bit complexity of the arithmetic operations of both GreedyInterval and Algorithm 1. We define the bit complexity as follows, following the definition of Grötschel, Lovász and Schrijver [15].

▶ **Definition 49** (Bit complexity). *Let $\langle v \rangle$ denote the bit complexity of value v.*

- *If v is an integer, then $\langle v \rangle = 1 + \lceil \log_2(|v| + 1) \rceil$, i.e. the number of bits used to encode v, including the sign bit.*
- *If v is a fraction $\frac{v^n}{v^d}$, then $\langle v \rangle = \langle v^n \rangle + \langle v^d \rangle$.*
- *If v is a point (v_x, v_y), then $\langle v \rangle = \langle v_x \rangle + \langle v_y \rangle$.*

Note that $\langle v \rangle$ denotes the number of bits used to represent the value v up to a constant factor, as it does not include bits used to separate, i.e., the numerator and denominator of a fraction. The notation is stronger than $\mathcal{O}$-notation, as it does not allow to hide a possible constant, to ensure that the hidden constant in the $\mathcal{O}$-notation truly is a constant. In the following lemmas, we will need the following rewriting properties of bit complexity on integers.

▶ **Lemma 50** (Properties of bit complexity). *Let a and b be integers. Then the following holds:*

- $\langle a \cdot b \rangle \leq \langle a \rangle + \langle b \rangle$.
- $\langle a + b \rangle \leq 1 + \max\{\langle a \rangle, \langle b \rangle\}$.

Proof. Follows from Definition 49 and logarithm rules. ◀

In this section we shall assume, that the input polygon P is encoded in N bits as a list of points, i.e., $4n$ integers encoded in binary. The polygon may be encoded using fewer bits by a different clever encoding, however, as the main goal is to show that contiguous art gallery problem is in the complexity class P, this simple encoding will suffice.

To bound the complexity of the points produced by GreedyInterval, we need the notion of a *scalar point*, which is defined as a point using two vertices of the input polygon and a fractional scalar, which denotes at what fractional point along the line defied by the two vertices the scalar point is located. Note that the scalar point does not need to be on the segment between the two points that defines it. Crucially, the intersection between two lines defined from three input vertices and a scalar point is a scalar point, as shown in the following lemma, which also bounds the bit complexity of the new scalar point as a function of the old.

▶ **Lemma 51** (Bit complexity of segment intersection). *Let N be the number of bits used to represent the input polygon. Let V, W, B, C, and D be input vertices, with $V = \left(\frac{V_x^n}{V_x^d}, \frac{V_y^n}{V_y^d}\right)$, and similarly for the other input vertices, where all values $v_x^n, v_x^d, v_y^n, v_y^d$ of the points are integers. Let s be a scalar $\frac{s^n}{s^d}$, with s^n and s^d being integers, where $\langle s \rangle$ may be of arbitrary complexity. Let A be the scalar point $sV + (1-s)W$. Assume $L(A, B)$ and $L(C, D)$ are non-parallel lines and let the point E be their intersection. Then there exists a scalar $t = \frac{t^n}{t^d}$, such that the intersection E is a scalar point of the form $tC + (1-t)D$, where $\max\{\langle t^n \rangle, \langle t^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.*

Proof. The setup is illustrated in Figure 39. Using a formula for the intersection of the lines defined by the segments $\overline{A\,B}$ and $\overline{C\,D}$ as stated by Goldman [13], the values of t^n and t^d

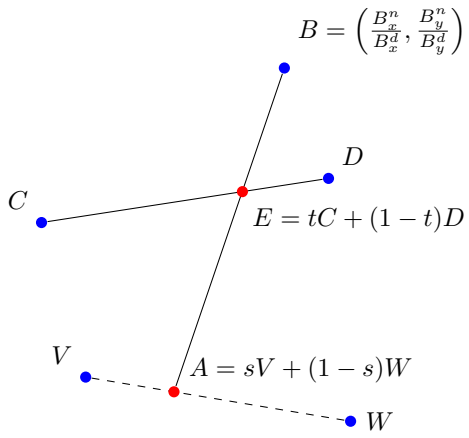


Figure 39 Setup of the intersection. Blue points denote input vertices and the red points denote scalar points. Note that it is not a requirement that the segments $\overline{A\,B}$ and $\overline{C\,D}$ intersect, only that their lines intersect.

can be expressed as follows:

$$
\begin{aligned}
t^n = \quad & s^n(+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d + B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n \\
& + B_x^d B_y^n C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^n W_x^d W_y^d \\
& - B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d - B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^n W_x^d W_y^d \\
& - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^d V_x^n V_y^d W_x^d W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^d W_y^n) \\
& + s^d(+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d + B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n \\
& + B_x^n B_y^d C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^d W_y^n - B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n \\
& - B_x^d B_y^n C_x^d C_y^d D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^d W_y^d)
\end{aligned}
$$

$$
\begin{aligned}
t^d = \quad & s^n(+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^n V_y^d W_x^d W_y^d + B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n \\
& + B_x^d B_y^d C_x^d C_y^n D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d + B_x^d B_y^d C_x^n C_y^d D_x^d D_y^d V_x^d V_y^n W_x^d W_y^d \\
& - B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d - B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^n W_x^d W_y^d \\
& - B_x^d B_y^d C_x^d C_y^n D_x^d D_y^d V_x^n V_y^d W_x^d W_y^d - B_x^d B_y^d C_x^n C_y^d D_x^d D_y^d V_x^d V_y^d W_x^d W_y^n) \\
& + s^d(+ B_x^d B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^n W_y^d + B_x^d B_y^d C_x^n C_y^d D_x^d D_y^d V_x^d V_y^d W_x^d W_y^n \\
& + B_x^d B_y^n C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^d + B_x^n B_y^d C_x^d C_y^n D_x^d D_y^d V_x^d V_y^d W_x^d W_y^d \\
& - B_x^d B_y^d C_x^d C_y^d D_x^n D_y^d V_x^d V_y^d W_x^d W_y^n - B_x^d B_y^d C_x^d C_y^n D_x^d D_y^d V_x^d V_y^d W_x^n W_y^d \\
& - B_x^d B_y^n C_x^n C_y^d D_x^d D_y^d V_x^d V_y^d W_x^d W_y^d - B_x^n B_y^d C_x^d C_y^d D_x^d D_y^n V_x^d V_y^d W_x^d W_y^d)
\end{aligned}
$$

Following Lemma 50 it holds that $\langle t^n \rangle$ and $\langle t^d \rangle$ both are bounded by $\max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$, as the point values of the input vertices are bounded trivially by N, and therefore $\max\{\langle t^n \rangle, \langle t^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$. ◀

Further, the bit complexity of a scalar point, when computed into a point, is bounded.

▶ **Lemma 52** (Bit complexity of scalar point). *Let N be the number of bits used to represent the input polygon. Let V and W be input vertices, with $V = \left(\frac{V_x^n}{V_x^d}, \frac{V_y^n}{V_y^d}\right)$, and similarly for W, where each component of the points are integers. Let s be a scalar $\frac{s^n}{s^d}$ of arbitrary*

complexity, with s^n and s^d being integers, and let A be the scalar point $sV + (1-s)W$. Then $\langle A \rangle \leq 4 \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.

Proof. Let $A = (A_x, A_y)$. It then holds that

$$A_X = \frac{V_x^d W_x^n s^d - V_x^d W_x^n s^n + V_x^n W_x^d s^n}{V_x^d W_x^d s^d},$$

and similarly for A_y. Applying Definition 49 and Lemma 50 immediately concludes the proof. ◀

Using the lemmas on scalar points and intersections, we can show that when the input point to GreedyInterval is a scalar point, then the output point is a scalar point, which scalar complexity is bounded by the scalar complexity of the input point.

▶ **Lemma 53** (Bound output bit complexity of GreedyInterval)**.** *Let P be a simple polygon encoded in N bits and let x be a scalar point on ∂P. Let V and W be vertices of P and s be a scalar $\frac{s^n}{s^d}$, s.t. $x = sV + (1-s)W$. Then there exists vertices V' and W' of P and a scalar $s' = \frac{s'^n}{s'^d}$, s.t. $G(x) = s'V' + (1-s')W'$, and it holds that $\max\{\langle s'^n \rangle, \langle s'^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.*

Proof. GreedyInterval proceeds in two phases. First, the feasible region F of x and a maximal number of contiguous edges from x is computed. Next, the guard seeing the furthest along the next edge of the boundary is then a corner of F, and $\mathrm{G}(x)$ is found. We therefore need to bound the complexity of the corners of F to then bound the final output point.

The feasible region F is found by the intersection of the visibility polygons from x and some of the input vertices. It must therefore hold that each vertex of F is the intersection of lines from the visibility polygons. Each line of a visibility polygon is either from a segment of P, which is therefore a line from a input vertex to a input vertex, a line from a reflex vertex to a input vertex, which is a input vertex to a input vertex line, or from the point x to a reflex vertex. Two lines both containing x must intersect in x, and such point is by definition a scalar point between input vertices. Any other intersection is then either between four input vertices, or three input vertices and the point x. By Lemma 51, it holds that each corner of the feasibility region F is a scalar point on the form $s''V'' + (1-s'')W''$, for some input vertices V'' and W'', and scalar $s'' = \frac{s''^n}{s''^d}$, and it holds that $\max\{\langle s''^n \rangle, \langle s''^d \rangle\} \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$.

Next, the point $\mathrm{G}(x)$ is found as the intersection of the line segment of some edge of P, and the line between a guard placement and a reflex vertex of P. As the guard placement is a corner of F, then the intersection is between three input vertices and a scalar point. Let the guard point be $s''V'' + (1-s'')W''$. By Lemma 51, then there are vertices V' and W', which is the endpoints of the edge $\mathrm{G}(x)$ is on, and a scalar $s' = \frac{s'^n}{s'^d}$, s.t. $\mathrm{G}(x) = s'V' + (1-s')W'$, and it holds that $\max\{\langle s'^n \rangle, \langle s'^d \rangle\} \leq \max\{\langle s''^n \rangle, \langle s''^d \rangle\} + \mathcal{O}(N) \leq \max\{\langle s^n \rangle, \langle s^d \rangle\} + \mathcal{O}(N)$. This therefore concludes the lemma. ◀

Similarly, it can be shown that the internal arithmetic operations GreedyInterval performs are bounded by the complexity of the input point.

▶ **Lemma 54** (Bound internal bit complexity of GreedyInterval)**.** *Let P be a simple polygon encoded in N bits and let x be a point on ∂P. Then the internal arithmetic operations of* GreedyInterval *with input x is polynomially bounded in complexity by N and $\langle x \rangle$.*

Proof. Each internal arithmetic computation of GreedyInterval are computed by a constant size straight line program, which therefore is computeable in time linear in N and $\langle x \rangle$. By Theorem 3 GreedyInterval performs polynomially many such computations in N, which concludes the lemma. ◀

We now have the building blocks necessary to show the main theorem.

▶ **Theorem 2.** *The contiguous art gallery problem for a simple polygon encoded in N bits is a member of the complexity class P.*

Proof. Algorithm 1 repeatably applies GreedyInterval to make revolutions around the polygon. By Lemma 54 the internal arithmetic operations of GreedyInterval are bounded by the complexity of the input point and the polygon. It therefore suffices to argue for the bit complexity of each intermediate point on ∂P where GreedyInterval is applied to.

Algorithm 1 starts at some point on ∂P. Let this starting point be some vertex V of P. Let W be any other vertex of P. It then holds that the starting point is on the form $sV + (1-s)W$ for scalar $s = 1/1$. Let $s_t = \frac{s_t^n}{s_t^d}$ be the scalar after t applications of GreedyInterval. By induction and Lemma 53 it holds that $\max\{\langle s_t^n \rangle, \langle s_t^d \rangle\} \leq \max\{\langle s_0^n \rangle, \langle s_0^d \rangle\} + t \cdot \mathcal{O}(N) = \mathcal{O}(tN)$. By Theorem 1 the number of applications of GreedyInterval is $\mathcal{O}\big(k^{*2} n^3\big)$, which bounds t. As each scalar s_t corresponds to a scalar point is between some input vertices of P, then by Lemma 52 it therefore holds that the bit complexity of any point computed by GreedyInterval on the boundary ∂P is bounded by $\mathcal{O}\big(k^{*2} n^3 \cdot N\big)$. This, by Lemma 54, concludes the proof. ◀

7 Open problems

We showed that the contiguous art gallery problem is solvable in $\mathcal{O}\big(k^* n^5 \log n\big)$ time by bounding the number of revolutions before Algorithm 1 finds an optimal solution to $\mathcal{O}\big(k^* n^3\big)$ and showing that the bit complexity is bounded. We conjecture that $\mathcal{O}(1)$ revolutions are sufficient. To provide evidence for this we simulated more than 2.000.000 random art galleries using the provided C++ implementation, and in all instances, it found an optimal solution (a repetition even) within 4 revolutions. If this conjecture is true, the complexity of Algorithm 1 becomes $\mathcal{O}\big(n^2 \log n\big)$. Analyzing the behavior of Algorithm 1 on axis-aligned input polygons appears to be a good step towards proving this conjecture.

We leave it as an intriguing open problem to decide whether the contiguous art gallery problem with holes is polynomial-time solvable, in contrast to the other art gallery variants we considered (see Table 1).

If an unrestricted guard may cover h intervals ($h > 1$), we believe the problem is NP-hard. This is the case for $h = n$ since this is exactly the edge-covering art gallery problem.

References

1 Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. The art gallery problem is $\exists\mathbb{R}$-complete. *J. ACM*, 69(1), December 2021. doi:10.1145/3486220.

2 Mikkel Abrahamsen, Joakim Blikstad, André Nusser, and Hanwen Zhang. Minimum star partitions of simple polygons in polynomial time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, page 904–910, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3618260.3649756.

3 Mikkel Abrahamsen and Bartosz Walczak. Common tangents of two disjoint polygons in linear time and constant workspace. *ACM Trans. Algorithms*, 15(1), December 2018. doi:10.1145/3284355.

4 Reymond Akpanya, Bastien Rivier, and Frederick B. Stock. Open problems CCCG 2024. In *Proceedings of the 36th Canadian Conference on Computational Geometry*, pages 167–170, 2024.

5 Avis and Toussaint. An optimal algorithm for determining the visibility of a polygon from an edge. *IEEE Transactions on Computers*, C-30(12):910–914, 1981. doi:10.1109/TC.1981.1675729.

6 Bentley and Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979. doi:10.1109/TC.1979.1675432.

7 Ahmad Biniaz, Anil Maheshwari, Magnus Christian Ring Merrild, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, Eliot W. Robson, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, Thomas Shermer, Jack Spalding-Jamieson, Rolf Svenning, and Da Wei Zheng. Polynomial-Time Algorithms for Contiguous Art Gallery and Related Problems. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry (SoCG 2025)*, volume 332 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:21, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SoCG.2025.20, doi:10.4230/LIPIcs.SoCG.2025.20.

8 Ahmad Biniaz, Anil Maheshwari, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, and Thomas Shermer. Contiguous boundary guarding, 2024. URL: https://arxiv.org/abs/2412.15053, arXiv:2412.15053.

9 Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer New York, 1998. doi:10.1007/978-1-4612-0701-6.

10 H El Gindy and D Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2(2):186–197, 1981. doi:10.1016/0196-6774(81)90019-5.

11 Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of cgal a computational geometry algorithms library. *Software: Practice and Experience*, 30(11):1167–1202, 2000.

12 Efi Fogel, Ophir Setter, Ron Wein, Guy Zucker, Baruch Zukerman, and Dan Halperin. 2D regularized boolean set-operations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgBooleanSetOperations2.

13 Ronald Goldman. Intersection of two lines in three-space. In ANDREW S. GLASSNER, editor, *Graphics Gems*, page 304. Morgan Kaufmann, San Diego, 1990. doi:10.1016/B978-0-08-050753-8.50064-4.

14 Günther Greiner and Kai Hormann. Efficient clipping of arbitrary polygons. *ACM Trans. Graph.*, 17(2):71–83, April 1998. doi:10.1145/274363.274364.

15 Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988. doi:10.1007/978-3-642-97881-4.

16 Michael Hemmer, Kan Huang, Francisc Bungiu, and Ning Xu. 2D visibility computation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgVisibility2.

17 J. Mark Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14(4):799–817, 1985. doi:10.1137/0214056.

18 J. Mark Keil and Jorg-R. Sack. Minimum decompositions of polygonal objects. In Godfried T. TOUSSAINT, editor, *Computational Geometry*, volume 2 of *Machine Intelligence and Pattern Recognition*, pages 197–216. North-Holland, 1985. doi:10.1016/B978-0-444-87806-9.50012-8.

19 Aldo Laurentini. Guarding the walls of an art gallery. *The Visual Computer*, 15(6):265–278, 1999. doi:10.1007/S003710050177.

20 C.C. Lee and D.T. Lee. On a circle-cover minimization problem. *Information Processing Letters*, 18(2):109–115, 1984. doi:10.1016/0020-0190(84)90033-4.

21 D. Lee and A. Lin. Computational complexity of art gallery problems. *IEEE Transactions on Information Theory*, 32(2):276–282, 1986. doi:10.1109/TIT.1986.1057165.

22 D. T. Lee and Arthur K. Lin. *Computational Complexity of Art Gallery Problems*, pages 303–309. Springer New York, New York, NY, 1990. doi:10.1007/978-1-4613-8997-2_23.

23 J. Mark Keil. Chapter 11 - polygon decomposition. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 491–518. North-Holland, Amsterdam, 2000. doi:10.1016/B978-044482537-7/50012-7.

24 Francisco Martínez, Antonio Jesús Rueda, and Francisco Ramón Feito. A new algorithm for computing boolean operations on polygons. *Computers & Geosciences*, 35(6):1177–1185, 2009. doi:10.1016/j.cageo.2008.08.009.

25 Magnus C. R. Merrild, Casper M. Rysgaard, Jens K. R. Schou, and Rolf Svenning. An Algorithm for the Contiguous Art Gallery Problem in C++. https://github.com/RolfSvenning/ContiguousArtGallery, 2024.

26 Joseph O'Rourke. *Art gallery theorems and algorithms*. Oxford University Press, Inc., USA, 1987.

27 Mark H Overmars. Dynamization of order decomposable set problems. *Journal of Algorithms*, 2(3):245–260, 1981. doi:10.1016/0196-6774(81)90025-0.

28 Eliot W. Robson, Jack Spalding-Jamieson, and Da Wei Zheng. The analytic arc cover problem and its applications to contiguous art gallery, polygon separation, and shape carving, 2024. URL: https://arxiv.org/abs/2412.15567, arXiv:2412.15567.

29 Marcus Schaefer and Daniel źTefankoviăź. Fixed points, nash equilibria, and the existential theory of the reals. *Theor. Comp. Sys.*, 60(2):172–193, February 2017. doi:10.1007/s00224-015-9662-0.

30 Thomas C. Shermer. Recent results in art galleries (geometry). *Proc. IEEE*, 80(9):1384–1399, 1992. doi:10.1109/5.163407.

31 Jack Stade. The Point-Boundary Art Gallery Problem Is $\exists\mathbb{R}$-Hard. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry (SoCG 2025)*, volume 332 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 74:1–74:23, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SoCG.2025.74, doi:10.4230/LIPIcs.SoCG.2025.74.

32 The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html.

33 Jorge Urrutia. Art gallery and illumination problems. *Handbook of Computational Geometry*, 12 2000. doi:10.1016/B978-044482537-7/50023-1.

34 Bala R. Vatti. A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63, July 1992. doi:10.1145/129902.129906.

35 Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. *SIGGRAPH Comput. Graph.*, 11(2):214–222, July 1977. doi:10.1145/965141.563896.

36 Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman. 2D arrangements. In *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgArrangementOnSurface2.

A Vertex restricted contiguous art gallery

In the following, we show how to efficiently compute an optimal solution to the contiguous art gallery problem, when the problem is vertex restricted, as mentioned in Section 1.2 and Table 1. There are two variants of this restriction, one is when the *intervals are restricted to vertices* and the other is when *guards are restricted to vertices*, which we cover in Appendix A.1 and A.2, respectively. We let for the following the input be a simple polygon P containing n vertices.

A.1 Intervals are restricted to vertices

If the contiguous interval seen by a guard is restricted to start and end at a vertex, the problem can be solved as follows. In a polygon of n vertices, there are $\Theta(n^2)$ contiguous intervals of the boundary, that is, from any vertex to any other. However, not all of these may be valid as there may not exist a guard that can see the whole interval. When all valid intervals have been found, the algorithm of Lee and Lee [20] can be used to find an optimal solution. Note that if a valid interval is contained entirely in another valid interval, then it can be pruned by Lemma 17.1, without removing optimality. For the valid intervals generated, we report only the intervals starting at a vertex and being maximal clockwise.

This is exactly equal to computing GreedyInterval from a starting vertex and restricting the final point to the last vertex on the computed interval. An iteration of GreedyInterval, when the interval contains e edges, runs in $\mathcal{O}(en \log n) = \mathcal{O}(n^2 \log n)$ time (Theorem 3), leading to total $\mathcal{O}(n^3 \log n)$ time used to compute the intervals. Computing the optimal solution of these intervals takes $\mathcal{O}(n)$ time, as the intervals are generated in sorted order, which leads to $\mathcal{O}(n^3 \log n)$ time in total.

Note that if the interval $[v_i, v_j]$ is computed from vertex v_i, then the interval starting at v_{i+1} must be able to reach at least v_j. Recomputing GreedyInterval for each vertex does not use this fact, and leads to the following optimization. If the visibility polygon of a vertex can be efficiently removed from the feasible polygon GreedyInterval used to determine the guard location and therefore also to decide whether an interval is maximal, then recomputation time can be optimized. The intersection algorithm is capable of intersecting both visibility polygons and feasible regions. Computing the feasible polygon of a set of visibility regions is, therefore, solvable by a simple divide-and-conquer algorithm.

The result of Overmars [27] show how to convert such static divide-and-conquer algorithms into a dynamic version, allowing for both insertion and deletion of visibility polygons from the set. The intersection algorithm takes two objects and computes their intersection. Let m be the total number of underlying visibility polygons on which the intersection is computed, with m_1 and m_2 visibility polygons contained in the two objects to intersect. The intersection is computed in $\mathcal{O}((n+h)\log n)$ time, with h describing the number of intersections between the two objects, due to an algorithm by Martínez, Rueda, and Feito [24], see Section 3.2. Note that intersections must be either some vertex of an input object intersecting the edge of the other input object, or a proper intersection between edges of the input objects. Every such proper intersection must lead to a vertex in the output object. The complexity of the output object is $\mathcal{O}(n)$ (Lemma 4), and as there similarly is $\mathcal{O}(n)$ and $\mathcal{O}(n)$ vertices in the input objects, then the number of total intersections $h \leq \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$. The running time of the intersection is therefore $\mathcal{O}(n \log n)$. This, by Overmars [27, Theorem 3.4], yields a data structure over m visibility polygons, allowing for both insertions and deletions of visibility polygons in $\mathcal{O}(n \log n \log m)$ time, while maintaining the feasible polygon over all current visibility polygons.

The optimized algorithm is then as follows. Start at any vertex v_i, and insert the visibility polygon of this vertex in the data structure that maintains the feasible region polygon. Then repeatably insert the visibility polygon of the next vertex until the feasible polygon becomes empty. Let this lastly inserted vertex be v_{j+1}. Then there must exist a guard that can see the interval $[v_i, v_j]$, which is outputted. Note that on an insertion, the visibility polygon of the vertex must be computed in $\mathcal{O}(n)$ time, using the algorithm by Gindy and Avis [10], see Section 3.1, to then be inserted. To compute the interval starting at v_{i+1}, then v_i is deleted from the data structure, and until the feasible polygon again becomes empty, vertices from v_{j+2} and onward are inserted. This procedure then generates the interval starting at v_{i+1}. This process continues to compute the maximal interval starting at every vertex. During execution of this procedure, the number of vertices in the data structure is at most n, and therefore $m \leq n$. Every vertex is inserted and deleted $\mathcal{O}(1)$ times. It takes $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$ total time to compute the visibility polygons of every vertex. The overall running time is therefore $\mathcal{O}(n^2) + n \cdot \mathcal{O}(1) \cdot \mathcal{O}(n \log n \log m) = \mathcal{O}(n^2 \log^2 n)$. This outputs n maximal intervals. Computing an optimal solution from the intervals takes $\mathcal{O}(n)$ time, as the intervals are generated in sorted order, leading to an overall $\mathcal{O}(n^2 \log^2 n)$ time to compute an optimal solution.

A.2 Guards are restricted to vertices

If the guard locations are restricted to the vertices, the problem can be solved as follows. If the polygon P has n vertices, then there is only n possible placements of a guard. Note that the visibility polygon from a point v yields exactly the area that can be seen from v. By intersecting the visibility polygon with the boundary ∂P, the contiguous intervals of ∂P can be computed. The visibility polygon from a vertex v_i can be computed in $\mathcal{O}(n)$ time by the algorithm of Gindy and Avis [10], see Section 3.1. The intersection with ∂P is computeable in $\mathcal{O}(n)$ time. Each guard may see multiple contiguous intervals of ∂P, however, there is at most $\mathcal{O}(n)$ intervals for each guard. In total, all contiguous intervals visible to a guard located in any vertex of P are computeable in $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$ time yielding $\mathcal{O}(n^2)$ intervals. The intervals are not computed in sorted order, so the Lee and Lee [20] algorithm computes an optimal solution in $\mathcal{O}(n^2 \log n)$.

B Proof of supporting lemmas

B.1 Feasible regions are connected

An important property of the feasible region F is that it is connected. To prove this, we show a generalization of Avis and Toussain [5, Lemma 1]:

▶ **Lemma 55** (Convex viewing lemma). *Let a, b and c be points in the interior of P (or on ∂P) such that a sees both b and c and that there is some point $d \in P$ on $\overline{bc}$, which is not visible from a. Then ∂P must intersect $\overline{bc}$.*

Proof. Since a can see both b and c, $\overline{ac}$ and $\overline{bc}$ are unobstructed. However, since a cannot see d, ∂P must intersect $\overline{ad}$. Since a, b and c are all inside P then ∂P must intersect triangle abc to enter, so it can block d from a, however, this can only happen by intersecting $\overline{bc}$ (see Figure 40). ◀

The feasible regions are constructed as the intersection between *visibility polygons* for single points on ∂P. We cover in detail how the GreedyInterval algorithm computes these in Section 3.

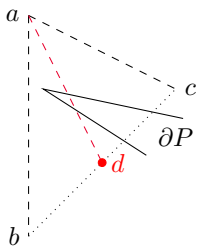


Figure 40 When a can see b and c, but not d, some part of ∂P intersects $\overline{b\,c}$.

A visibility polygon from a point $a \in \partial P$ can be decomposed into *visible triangles* by drawing lines from a to all the vertices of ∂P visible from a (see Figure 41).

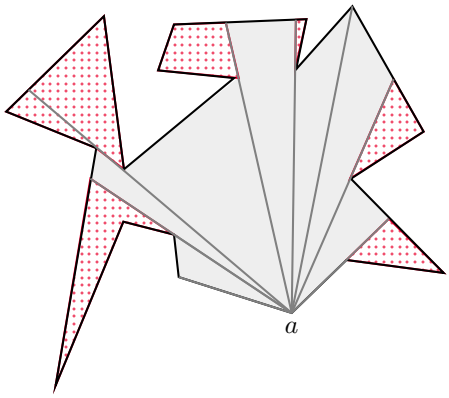


Figure 41 The grey region is a visibility polygon $VP(P, a)$ from point a in polygon P. It is further subdivided into visible triangles and the red non-visible areas are pockets that connect to $VP(P, a)$ at windows.

Note that these visible triangles may be degenerate (i.e. just a line). The areas marked in red, which are not visible to a, we denote *pockets*. Each pocket is connected to the visibility polygon through line segments shared with one visible triangle. We denote these shared line segments as *windows* (similar to Gindy and Avis [10]). In Figure 41 each pocket has exactly one window. The following lemma shows that this is always true:

▶ **Lemma 56** (Pockets have exactly one window)**.** *Let $a \in \partial P$ and let R be a pocket of $VP(P, a)$. Then there is exactly one window for R.*

Proof. We show the lemma by contradiction. It is clear, that R must be connected to at least one visible triangle by an edge, as R otherwise would be disconnected from P and hence not a part of P.

Thus we assume there are at least 2 windows for R. Consider the connected components of $\partial P \cap \partial R$, i.e. the boundary of R without the windows. As parts of ∂P they are connected in a cycle with other parts of ∂P and as parts of ∂R they are connected in a cycle by windows. We consider a point b, which is a vertex of R, where one edge connected to b is in a component of $\partial P \cap \partial R$, which we denote f and the other is a window w_b. Furthermore, we assume B is the closer point to a, of the endpoints of w_b. Let c be the other endpoint f. See Figure 42.

We now consider where ∂P can continue from b. We consider four cases:

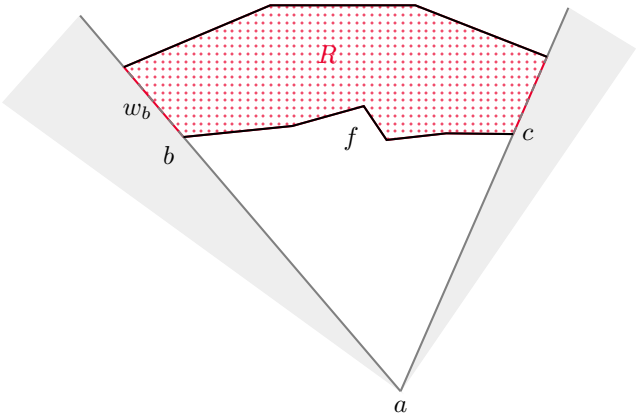


Figure 42 When a pocket is connected by multiple windows, the polygon cannot be simple.

▷ Case 1. ∂P can continue along w_b. However, this will contradict the assumption that b is connected to the window.

▷ Case 2. ∂P can continue into the visible triangle, but then the triangle will no longer be completely visible, a contradiction.

▷ Case 3. ∂P can continue into R, however R is completely contained in P, and if an edge of ∂P lies in R, there will be some area on one side of the edge, which is not contained in P. Thus, we have a contradiction.

▷ Case 4. ∂P can continue along $\overline{a\,b}$ or the white region below R (see Figure 42). Once it enters here, it will become stuck inside the area bounded by the two visible triangles and f. It can exit through C, but this will create a loop in ∂P not including the top of R, which would introduce a hole in P.

It could also enter a. However, we could do the same case analysis for C, and it also need to enter into a, thus we again have a loop, and we are done. ◀

With this, we are now ready to prove the main lemma of this subsection:

▶ **Lemma 39.** *For any $a, b \in \partial P$, the feasible region $F([a,b])$ is connected. Furthermore, for any convex subset $C \subset P$, $C \cap F([a,b])$ is convex.*

Proof. The proof of connectivity follows by induction in the number of visible polygons we intersect in the GreedyInterval algorithm (see Section 3).

For the base case, i.e. $F([a,a])$ where $a \in \partial P$, the feasible region is the visibility polygon $VP(P,a)$. Visibility polygons are star-shaped, hence connected.

For the induction step, we assume the feasible region seeing $[a, v_i]$ is connected and let v_{i+1} be the next vertex along ∂P or the point b if no such vertex exits. We assume for contradiction $F([a, v_{i+1}])$ is not connected. For this to happen, the feasible region $F([a, v_i])$ has to enter a pocket of the visibility polygon somewhere and exit elsewhere, so that the visibility polygon contains two disconnected components of $F([a, v_i])$ (see Figure 43). Since each pocket only has one window by Lemma 56, the two components will intersect the same visible triangle. Let points s and t from different components of $F([a, v_i])$ in the same visible triangle.

Since the previous feasible region is the intersection of visibility polygons for points $[a, v_i]$, there must be some point for which s and t are visible while some point on $\overline{s\,t}$ is not. By

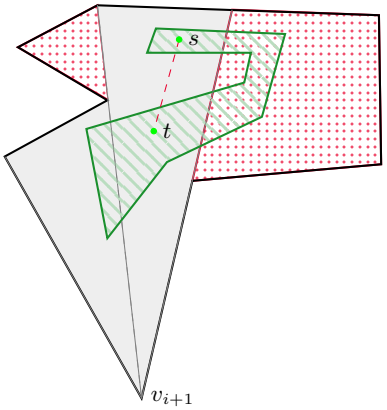


Figure 43 In the induction step, we intersect $VP(P, v_{i+1})$ (in gray) with the feasible region $F([a, v_i])$ (in green) and assume for contradiction that this disconnects the feasible region. Taking points s and t in different components, we now use convex viewing lemma (Lemma 55) to get a contradiction

Lemma 55, there must be some part of ∂P that intersects $\overline{s\,t}$, however, $\overline{s\,t}$ is contained in a visible triangle, where no part of ∂P can lie. Hence, the new intersected feasible region will continue to be connected, finishing the induction.

The same considerations about s and t show the convexity claim. ◀

B.2 Visible intervals cannot block their endpoints

Since we only look at blockage (Remark 21), it will be important to look at the point that blocks parts of edges from guards:

▶ **Lemma 57** (What g sees around blocking vertices). *Let g be a guard seeing $[y, z]$ and assume that c is a vertex of P that blocks g from seeing more than z. Then either g cannot see both edges connected to c, or one of these edges ℓ is a segment of $L(z, g)$ and g cannot see both of the edges connected to ℓ.*

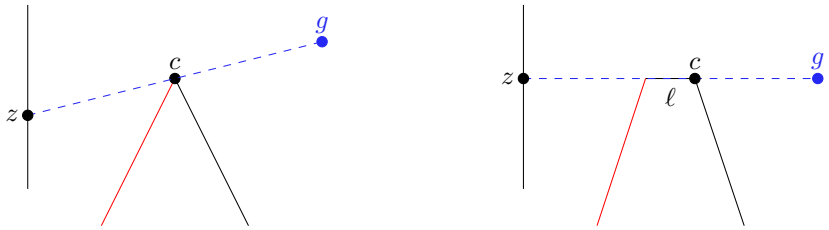


Figure 44 Left, c blocks g and g sees only one edge connected to c. Right, g sees both edges, but one is a segment of $L(g, c)$ and the edge after is not visible.

Proof. Assume g can see both edges connected to c. Since c blocks g from seeing any more than z, we know z, g and c are collinear. Let a and b be points on the edge after, respectively before c, which are visible from c. We now show, that either a and b lie on different sides of $L(g, c)$, or one of the points lies on $L(g, c)$.

So assume for contradiction a and b lie on the same side of $L(g, c)$. Assume w.l.o.g. $\angle cga \leq \angle cgb$.

Thus the ray $\overrightarrow{g\,a}$ intersects $\overline{b\,c}$, at some point which we denote d. Let m be the midpoint of $\overline{a\,c}$ and q the midpoint of $\overline{c\,d}$. We split into two cases: Either a lies on $\overline{g\,d}$ or d lies on $\overline{g\,a}$ (see Figure 45).

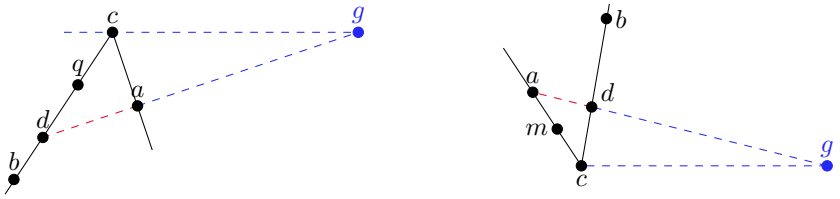


Figure 45 a and b on the same side of $L(g, c)$.

▷ Case 1. If a lies on $\overline{g\,d}$ then g cannot see q (Figure 45 (left)).

▷ Case 2. If d lies on $\overline{g\,a}$ then g cannot see m (Figure 45 (right)).

Since we assumed that g sees a, b, and c, g must also be able to see $\overline{a\,c}$ and $\overline{b\,c}$, which is impossible as either q or m cannot be seen, thus a and b must either lie on different sides of $L(g, c)$ or one must lie on $L(g, c)$ (we cannot have both lie on $L(g, c)$ since c is a vertex).

Assuming a and b are on either side of $L(g, c)$, now the interval $[a, b]$ will block the view from g to z completely, thus we can discard this case. We assume w.l.o.g. a lies on $L(g, c)$. Let the endpoint of the edge containing a different from c be j and let m be the other edge connected to j. If g can see no more of m than j, we are done, so assume g can see some point u on m. u cannot lie on $L(g, c) = L(g, j)$ since j is a vertex. If u and b lie on the same side of $L(g, c)$ we have the same problem as for a and b above (Figure 46 left). If k and b lie on different sides of $L(g, c)$ then $[b, u]$ will block the view from g to z (Figure 46). ◀

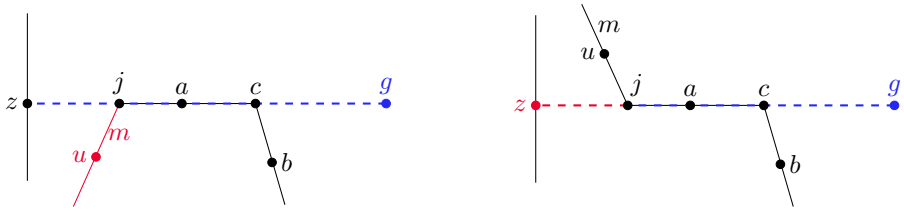


Figure 46 Left, u lies on same side as b and is not visible from g. Right, u lies on different side, but now z is not visible from g.

This can now be used to show where the blocking points are located in the figure:

▶ **Lemma 40** (Blockings happen outside visible area). *Let $y \in \partial P$ and $z = G(y)$ and assume that neither y nor z is a vertex of P. Let g be a guard seeing $[y, z]$ and let c be a vertex blocking the view from g to z. Then $c \notin [y, z]$.*

Proof. Assume for contradiction that $C \in [y, z]$. Since c is a vertex of P, c is neither y nor z, thus y and z must lie on different sides of c along $[y, z]$. From Lemma 57 we have two cases for what g can see close to c along ∂P.

▷ Case 1. If g cannot see both edges connected to c, either $[y, c]$ or $[c, z]$ is not visible to g contradicting the fact that g sees $[y, z]$.

▷ Case 2. If g is able to see both edges connected to c, then we know one is contained in $L(g, z)$ and the next edge is not visible from g, hence z must lie on the edge connected to c, which is contained in $L(g, z)$, for $[c, z]$ to be visible. However now z can be moved until the endpoint of the edge containing z, hence z is a vertex, which is contradicting the assumption (see Figure 47). ◀

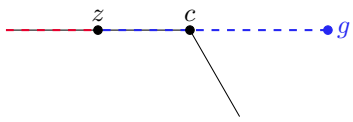


Figure 47 If c is to act as blockage between g and z, then g should see no more than z. However, when g, c and z are collinear g can see the entire edge containing z.

C Insightful examples

C.1 Example of two combinatorially indistinguishable functions

▶ **Example 58.** An examples of functions, which combinatorially look identically, but behave very different is given below:

For this, we represent points on the circle by numbers in $[0, 1)$. We then consider the two functions G_1 and G_2. The first (see Figure 48 (left)) is defined as $G_1(x) = \{x + \frac{1}{k} - \frac{\varepsilon}{k}\}$ where ε is some fixed small number and $\{a\}$ is the decimal part of a (i.e. $a = \lfloor a \rfloor + \{a\}$). This will require $k + 1$ guards, but ε^{-1} steps of GreedyInterval will be required to satisfy one of the optimality conditions (in this case we satisfy Corollary 31).

Secondly, we define $G_2(x) = \{x + \frac{1}{k} - \alpha\frac{\{kx\}}{k}\}$, where α is some number in $(0, 1)$ (see Figure 48 (right)). It has an optimal solution at $y_i = i/k$ with k intervals. However starting anywhere other than in an optimal solution and running Algorithm 1 will never yield an optimal solution, since $\frac{\{xk\}}{k}$ is the distance x needs to move backwards to hit the optimal solution, however we only move α times that distance each step.

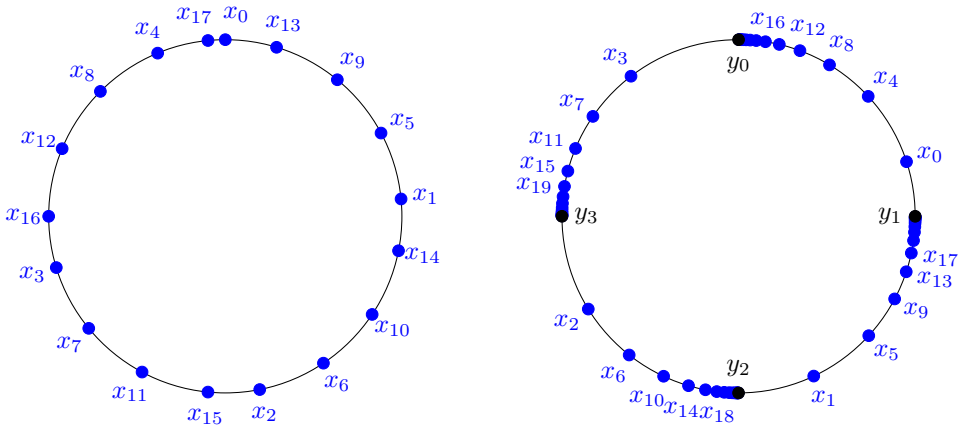


Figure 48 Left, G_1 with $k = 4$ is used to generate a greedy sequence, where x_{17} shows, that we are optimal. Right, G_2 with $k = 4$ and $\alpha = 0.1$ is used. Starting at $x_0 = 0.2$, we never see an optimal solution.

Combinatorially, these two functions are identical (until ε^{-1} rounds have passed), so we have no guarantee, that the algorithm terminates with a solution in polynomial time.

C.2 Polygons with holes

One natural generalization of the contiguous art gallery problem is the contiguous art gallery problem with holes. In this section, we study the variant in which the goal is to guard the external boundary of the polygon.

Many parts of the geometric analysis break down when we introduce holes. The most glaring is the fact that the entire strategy of Proposition 48 does not work, as we are using the fact that the pivot points (and edges close to them) have to be guarded by a guard at some point and now these pivot points could be on a hole.

Furthermore, the algorithm given for GreedyInterval in Section 3 does not work as a point in P can now see two vertices of P without seeing the entire edge between them. This issue can be fixed, but even if it is, we show that Algorithm 1 can run in superpolynomial time in the number of vertices when P has holes.

▶ **Example 59** (Octagon with hole)**.** Consider a regular octagon with a rotated regular octagon inside like shown on Figure 49, where the inner octagons edges line up exactly with the dashed line segments. Placing guards g_1 and g_2 will guard the entire boundary contiguously.

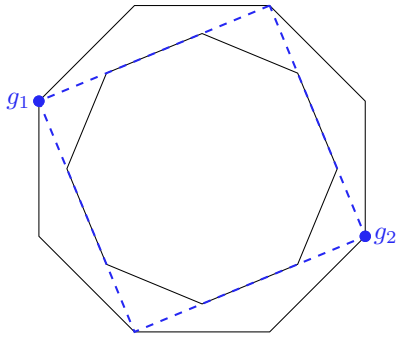


Figure 49 P is an octagon with an octagonal hole in the center. Guards at g_1 and g_2 will guard the entire boundary contiguously.

Now we enlarge the inner polygon by a tiny ε, which will make this solution invalid. When we run Algorithm 1 in this slightly different polygon, it is evident, that the best guard is placed on the outer boundary of P as far as the starting point can see (see Figure 50) and the greedy interval is found by taking the furthest point which this guard can see.

As long as the found endpoint/guard is closer to the next vertex than the previous vertex (along the outer boundary of P), it is the same vertex of the inner polygon which will block the view to the next guard/endpoint (the one marked in Figure 51).

We consider how far a point on the lower edge of P can see on the left vertical edge of P. To do this, we embed P into a coordinate system with the lower edge having endpoints in $(0, 0)$ and $(1, 0)$. The relevant coordinates are drawn on Figure 51.

Calculating, we get that the equation of the dashed line is:

$$y = \left(\frac{\sqrt{2} + 2\delta}{\sqrt{2} - 1 + 2\varepsilon + 2\delta} - 1 \right) (\delta - x)$$

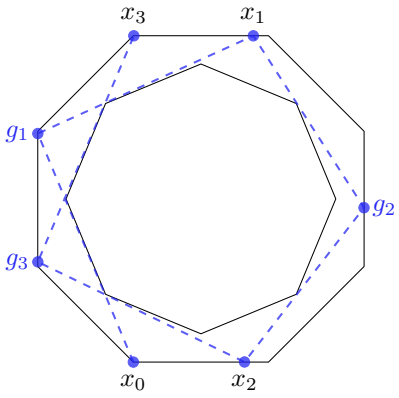


Figure 50 We have here enlarged the inner polygon with $\varepsilon = 0.001$, we now start Algorithm 1 at x_0. The optimal guard for x_0 is found by finding the furthest point along the outer boundary which x_0 can see. This point is denoted g_1 which can see until x_1 and so on.

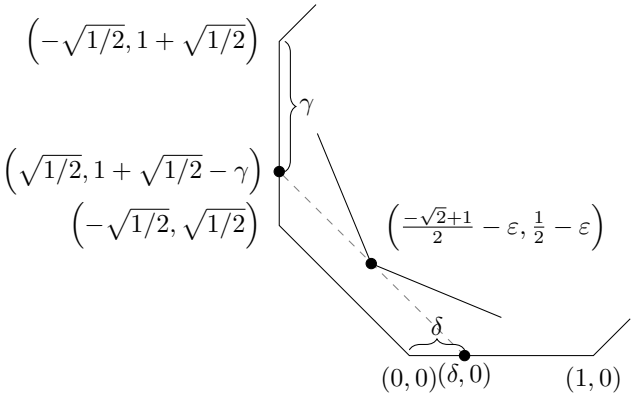


Figure 51 P embedded into a coordinate system with coordinates marked. A guard placed in $(\delta, 0)$ can see no longer than $(-\sqrt{1/2}, 1+\sqrt{1/2}-\gamma)$. Here γ is the distance between the new point and the next vertex of P.

Inserting $x = -\sqrt{1/2}$ in this equation will yield the y-coordinate, y', of the intersection with the dashed line and the left vertical edge:

$$\begin{aligned} y' &= \left(\frac{\sqrt{2}+2\delta}{\sqrt{2}-1+2\varepsilon+2\delta} - 1\right)\left(\delta + \sqrt{\frac{1}{2}}\right) \\ &= \frac{1+2\sqrt{2}\delta+2\delta^2}{\sqrt{2}-1+2\varepsilon+2\delta} - \sqrt{\frac{1}{2}} - \delta \end{aligned}$$

And now γ is calculated as $\gamma = 1 + \sqrt{\frac{1}{2}} - y'$:

$$\begin{aligned} \gamma &= \delta + \frac{2\delta - 2\delta^2 + (2+2\sqrt{2})\varepsilon}{\sqrt{2}-1+2\varepsilon+2\delta} \\ &\in (\delta, 7\delta + 12\varepsilon) \end{aligned}$$

We now chooce $\varepsilon = \frac{1}{2 \cdot 12 \cdot 8^{2N}}$ where N is some large number. Consider the sequence $(\gamma_i)_{i=0}^{2N}$ of distances from the furthest visible point and the next vertex when taking such

steps. Note that in Figure 51 we have, say, $\delta = \gamma_i$ and $\gamma = \gamma_{i+1}$, i.e. the δ is the member of the of the γ_i sequence that precedes γ. The above bound then becomes $\gamma_{i+1} \in (\gamma_i, 7\gamma_i + 12\varepsilon)$ and especially $\gamma_i < \gamma_{i+1}$ hence the associated local greedy sequences will have negative fingerprints and no repetitions.

Furthermore we show by induction that $\gamma_i < \frac{1}{2 \cdot 8^{2N-i}}$ as $\gamma_0 = 0$ and if $\gamma_i < \frac{1}{2 \cdot 8^{2N-i}}$, we get:

$$\begin{aligned}\gamma_{i+1} &< 7\gamma_i + 12\varepsilon \\ &= \frac{7}{2 \cdot 8^{2N-i}} + \frac{1}{2 \cdot 8^{2N}} \\ &\leq \frac{1}{2 \cdot 8^{2N-(i+1)}}\end{aligned}$$

And we thus have $\gamma_i \in [0, 1)$ for all $\gamma = 0, 1, \ldots, 2N$, thus for all the guards will be on the same two edges, i.e. we get no edge jumps. Thus in the first N greedy steps we do not reach a geometric progress condition.

Since N is independent of n the runtime of Algorithm 1 is unbounded in the real RAM model.

Chapter 10

Fast Area-Weighted Peeling of Convex Hulls for Outlier Detection

Fast Area-Weighted Peeling of Convex Hulls for Outlier Detection*

Vinesh Sridhar† Rolf Svenning‡

Abstract

We present a novel 2D convex hull peeling algorithm for outlier detection, which repeatedly removes the point on the hull that decreases the hull's area the most. To find k outliers among n points, one simply peels k points. The algorithm is an efficient *heuristic* for *exact* methods, which find the k points whose removal together results in the smallest convex hull. Our algorithm runs in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(n)$ space for any choice of k. This is a significant speedup compared to the fastest exact algorithms, which run in $\mathcal{O}\big(n^2 \log n + (n-k)^3\big)$ time using $\mathcal{O}\big(n \log n + (n-k)^3\big)$ space by Eppstein et al. [12, 14], and $\mathcal{O}\Big(n \log n + \binom{4k}{2k}(3k)^k n\Big)$ time by Atanassov et al. [4]. Existing heuristic peeling approaches are not area-based. Instead, an approach by Harsh et al. [17] repeatedly removes the point furthest from the mean using various distance metrics and runs in $\mathcal{O}(n \log n + kn)$ time. Other approaches greedily peel one convex layer at a time [20, 2, 19, 30], which is efficient when using an $\mathcal{O}(n \log n)$ time algorithm by Chazelle [7] to compute the convex layers. However, in many cases this fails to recover outliers. For most values of n and k, our approach is the fastest and first practical choice for finding outliers based on minimizing the area of the convex hull. Our algorithm also generalizes to other objectives such as perimeter.

1 Introduction

When performing data analysis, a critical first step is to identify outliers in the data. This has applications in data exploration, clustering, and statistical analysis [31, 9, 23]. Typical methods of outlier detection such as Grubbs' test [15] are based in statistics and require strong assumptions about the distribution from which the sample is taken. These are known as parametric outlier detection tests. If the sample size is too small or the distribution assumptions are incorrect, parametric tests can produce misleading results. For these reasons, non-parametric complementary approaches based in computation geometry have emerged. Our work follows this line of research and is based on the fundamental notion of a convex hull. For a set of points P, the convex hull is the smallest convex set containing P [10].

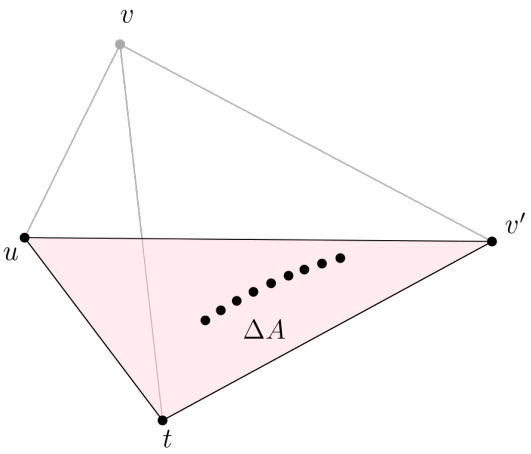


Figure 1: Here point v was peeled from the convex hull and replaced by v'. The previous triangle $\triangle tuv$ for u contained no points. However, when u's triangle becomes $\triangle tuv'$, the set of points ΔA affect the sensitivity $\sigma(u)$ of u. The size of ΔA may be $\Omega(n)$.

There are numerous definitions of outliers [22, 28, 3], but a general theme is that points without many close neighbors are likely to be outliers. As such, these outlying points tend to have a large effect on the shape of the convex hull. Prior work has applied this insight in different ways to identify possible outliers, such as removing points from the convex hull to minimize its diameter [1, 13], its perimeter [11], or its area [14, 12]. Motivated by the last category, we will consider likely outliers to be points whose removal causes the area of the convex hull to shrink the most. We propose a greedy algorithm that repeatedly removes the point $p \in P$ such that the area of P's convex hull decreases the most. We call the amount the area would decrease if point p is removed its *sensitivity* $\sigma(p)$. The removed point is guaranteed to be on the convex hull, and such an algorithm is known as a convex hull *peeling* algorithm [19, 30]. To find k outliers, we peel k points. Our algorithm is conceptually simple, though it relies on the black-box use of a dynamic (or deletion-only) convex hull data structure [18, 6]. We assume that points are in general position. This assumption may be lifted using perturbation methods [25].

*This work is supported in part by Independent Research Fund Denmark grant 9131-00113B and a fellowship from the Department of Computer Science at UC Irvine.

†University of California, Irvine, vineshs1@uci.edu

‡The Department of Computer Science, Aarhus University, rolfsvenning@cs.au.dk

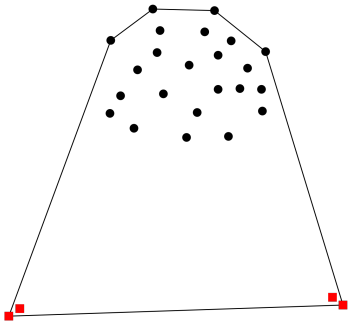


Figure 2: This figure demonstrates the limitations of our heuristic weighted-peeling approach. Clearly, the red squares are outliers, but because there are two squares close-by, the sensitivity of the red squares is minimal. Thus, our algorithm may peel all the valid points before peeling the outlier squares. Note that two k-peels for $k = 2$ would be sufficient to remove all outliers.

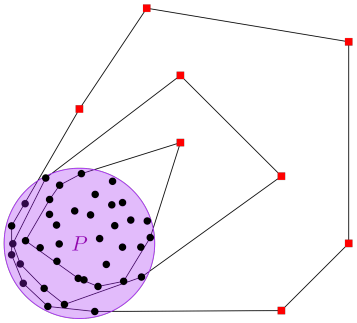


Figure 3: This example shows points drawn uniformly from a target disk P. Clearly, the outliers are the points marked as red squares. It shows the downside of peeling based on depth since many points have to be peeled before reaching the outliers on the second and third layers. In particular, if there are n points drawn uniformly from P, then its convex hull has expected size $\mathcal{O}(n^{1/3})$ [16].

The main challenge is maintaining the sensitivities as points are peeled. When peeling a single point v, there may be $\Omega(n)$ new points affecting the sensitivity $\sigma(u)$ for a different point $u \neq v$, as in Figure 1. In that case, naively computing the new sensitivity $\sigma(u)$ would take $\Omega(n)$ time. Nevertheless, we show that our algorithm runs in $\mathcal{O}(n \log n)$ time for any $1 \leq k \leq n$.

2 Related work

The two existing approaches for finding outliers based on the area of the convex hull took a more ideal approach. They considered finding the k points (outliers) whose removal together causes the area of the convex hull to decrease the most. We call this a *k-peel* and note that it always yields an area smaller or equal to that of performing k individual 1-peels. It is not hard to come up with examples where the difference in area between the two approaches is arbitrarily large such as in Figure 2. Still, these examples are quite artificial and require that outliers have at least one other point close by. More importantly, these methods are combinatorial in nature, and much less efficient than our algorithm. The state-of-the-art algorithms for performing a *k-peel* run in $\mathcal{O}(n^2 \log n + (n-k)^3)$ time and $\mathcal{O}(n \log n + (n-k)^3)$ space by Eppstein [12, 14] and $\mathcal{O}\left(n \log n + \binom{4k}{2k}(3k)^k n\right)$ time by Atanassov et al. [4]. While excellent theoretical results, for most values of $1 \leq k \leq n$ and n, the running time of both of these algorithms is prohibitive for practical purposes. Our contribution is a fast and practical heuristic for these ideal approaches. There are also several results for finding the k points minimizing other objectives such as the minimum diameter, perimeter, or area-enclosing rectangle [13, 29].

Another convex hull peeling algorithm is presented in [17]. Unlike in area-based peeling, they repeatedly remove the point furthest from the mean under various distance metrics. Letting d be the time to compute the distance between two points, their algorithm runs in $\mathcal{O}(n \log n + knd)$ time, which is also significantly slower than our algorithm for most values of k. Since they maintain the mean of the remaining points during the peeling process, each peel takes $\Theta(n)$ time.

Some depth-based outlier detection methods also use convex hulls. They compute a point set's convex layers, which can be defined by iteratively computing $P \setminus CH(P)$ and are computable in $\mathcal{O}(n \log n)$ time [7]. Here, points are deleted from the outermost-layer-in [20, 2, 19, 30]. While efficient, the natural example in Figure 3 is a bad instance for this approach.

3 Results

The main result of our paper is Theorem 1, that there exists an algorithm for efficiently performing area-weighted-peeling.

Theorem 1 *Given n points in 2D, Algorithm 1 performs area-weighted-peeling, repeatedly removing the point from the convex hull which causes its area to decrease the most, in $\mathcal{O}(n \log n)$ time.*

To prove Theorem 1, we derive Theorem 5, which bounds the total number of times points become *active* in any 2D convex hull peeling process to $\mathcal{O}(n)$.

Definition 3.1 (Active Points) *Let (t, u, v) be consecutive points on the first layer in clockwise order. A point p is active for u if, upon deleting u and restoring the first and second layers, p moves to the first layer.*

Intuitively, the active points are the points not on the convex hull that affect the sensitivities. Note that the active points form a subset of the points on the second convex layer. We define $A(u)$ to be the set of active points for point u in a given configuration. Furthermore, all points in $A(u)$ can be found by performing gift-wrapping starting from u's counterclockwise neighbor t while ignoring u. We use this ordering for the points in $A(u)$. In Theorem 7, we show that our algorithm generalizes to other objectives such as *perimeter* where the sensitivity only depends on the points on the first layer and the active points.

4 Machinery

In this section, we describe some of the existing techniques we use. To efficiently calculate how much the hull shrinks when a point is peeled, we perform tangent queries from the neighbours of the peeled point to the second convex layer. The tangents from a point q to a convex polygon L can be found in $\mathcal{O}(\log n)$ time both with [27] and without [21] a line separating q and L. In our application, such a separating line is always available, and either approach can be used. Tangent queries require that L is represented as an array or a balanced binary search tree of its vertices ordered (cyclically) as they appear on the perimeter of L. To allow efficient updates to L we use a binary tree representation that is leaf-linked such that given a pointer to a vertex its successor/predecessor can be found in $\mathcal{O}(1)$ time.

The convex layers of n points can be computed in $\mathcal{O}(n \log n)$ time using an algorithm by Chazelle [7]. Given l convex layers, after a single peel they can be restored in $\mathcal{O}(l \log n)$ time (Lemma 3.3 [24]). However, for our purposes we only need the 2 outermost layers for area calculations. As such, we explicitly maintain the two outermost layers L^1 and L^2, and we store all remaining points $P \backslash \{L^1 \cup L^2\}$ in a *center* convex hull. To restore L^1 we use tangent queries on L^2 as in [24]. To restore L^2 we use extreme point queries on the center convex hull which we maintain using a semi-dynamic [18] or fully-dynamic [6] convex hull data structures supporting extreme point queries in worst case $\mathcal{O}(\log n)$ time and updates in amortized $\mathcal{O}(\log n)$ time.

5 Area-Weighted-Peeling Algorithm

In this section, we describe Algorithm 1 in detail and show that its running time is $\mathcal{O}(n \log n)$.

At a high level, we want to repeatedly identify and remove the point which causes the area of the convex hull to decrease the most. Such an iteration is a *peel*, and we call the amount the area would decrease if point u was peeled the sensitivity $\sigma(u)$ of u. To efficiently find the point to peel, we maintain a priority queue Q

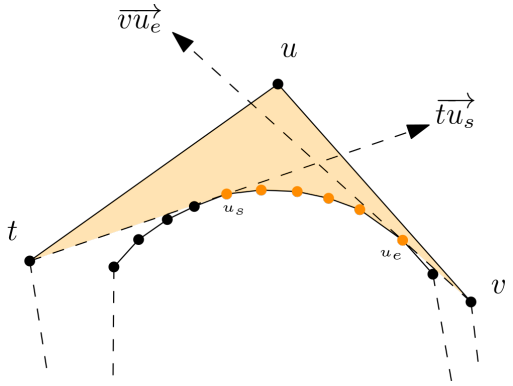


Figure 4: Using u's neighbors, we can perform two tangent queries on L^2 to recover the first and last active point of u, labeled u_s and u_e respectively, in $\mathcal{O}(\log n)$ time. Because we represent L^2 as a leaf-linked tree, we can walk along L^2 to recover all points of $A(u)$. The shaded part of the figure represents $\sigma(u)$.

on the sensitivities of hull points. Only points on the convex hull may have positive sensitivity, and in lines 2-6 we compute the initial sensitives of the points on the convex hull and store them in Q. For a hull point u, to compute its sensitivity $\sigma(u)$ we find its active points $A(u)$. Note they must be on the second convex layer, and if u's neighbors are t and v, then the points $A(u)$ are in the triangle $\triangle tuv$. In line 1 we compute the two outer convex hull layers represented as balanced binary trees. That allows us to compute $A(u)$ using tangent queries on the inner layer from t and v. Then $\sigma(u)$ can be found by computing the area of the polygon $\pentagon(t \circ v \circ A(u))$.

As points are peeled (lines 8-17) layers L^1 and L^2 must be restored. To restore L^1 when point u is peeled (line 9) we perform tangent queries on L^2 as in [24] to find u's active points $A(u)$ (line 10) and move $A(u)$ from L^2 to L^1. See Figure 4 for an example of tangent queries from L^1 to L^2.

To restore the broken part of L^2, we perform extreme point queries on the remaining points efficiently using a dynamic convex hull data structure D_{CH} (line 7) as in [18] or [6]. As described in Lemma 6, $A(u)$ is always contiguous on L^2. Therefore, removing $A(u)$ from L^2 requires us to restore it between two "endpoints" a and b. The first extreme point query uses line $\overline{ab}$ in the direction of u. If a point z from D_{CH} is found then at least two more queries are performed with lines $\overline{za}$ and $\overline{zb}$. In general, if k points are found then the number of queries is $2k + 1$. The k points are deleted from D_{CH}. This all happens on line 11.

Next, we compute the sensitivities of the new points on the hull (line 14) and insert them into the priority queue. Finally, we update the sensitivities of u's neighbors t and v (line 17), which, by Lemma 2(4), are the only two points already in Q whose sensitivity changes.

Algorithm 1: Weighted peeling

Input: A set of n points P in 2D

1 $L^1, L^2 \longleftarrow$ the first two convex layers of P
2 $Q \longleftarrow$ empty max priority queue
3 **for** $i = 1$ **to** $\left|L^1\right|$ **do**
4 $\quad u \longleftarrow L^1_i$
5 $\quad$ Compute sensitivity $\sigma(u)$ for u
6 $\quad Q.insert(u, \sigma(u))$
7 $D_{CH} \longleftarrow$ a dynamic convex hull data structure on $P \setminus \left\{L^1 \cup L^2\right\}$
8 **for** $i = 1$ **to** n **do**
9 $\quad u \longleftarrow Q.extractMax$
10 $\quad A(u) \longleftarrow u$'s active points
11 $\quad$ Delete u from L^1 and update L^1, L^2 and D_{CH}
12 $\quad$ **for** $i = 1$ **to** $|A(u)|$ **do**
13 $\quad\quad \bar{u} \longleftarrow A(u)_i$
14 $\quad\quad$ Compute sensitivity $\sigma(\bar{u})$ for $\bar{u}$
15 $\quad\quad Q.insert(\bar{u}, \sigma(\bar{u}))$
16 $\quad t, v \longleftarrow$ neighbors of u in L^1
17 $\quad$ Update $Q[t]$ and $Q[v]$

5.1 Analysis

The hardest part of the analysis is showing that the overall time spent on lines 14 and 17 is $\mathcal{O}(n \log n)$. We first show that, excluding the time spent on these lines, the running time of Algorithm 1 is $\mathcal{O}(n \log n)$. In line 1 we compute the first and second convex layers in $\mathcal{O}(n \log n)$ time by running any optimal convex hull algorithm twice. In lines 2 to 6, we compute the initial sensitivities by finding the points active for each $u \in L^1$. As described above, we can do this by applying two tangent queries, allowing us to recover the first and last extreme point for u. We can walk along L^2 between them to recover $A(u)$. Once $A(u)$ is found for each u, we find $\sigma(u)$ by computing the area of the polygon $\circlearrowleft(t \circ u \circ v \circ A(u))$, where t and v are u's neighbors. By Lemma 2(1), in this initial configuration each point on L^2 is active in at most three triangles. Thus, we make in total $\mathcal{O}\left(\left|L^2\right|\right) = \mathcal{O}(n)$ tangent queries, each of which costs $\mathcal{O}(\log n)$ time. Since the area of a simple polygon can be computed in linear time [26], all the area computations take $\sum_{u \in L^1} \Theta(1 + |A(u)|) = \mathcal{O}\left(\left|L^1\right| + \left|L^2\right|\right) = \mathcal{O}(n)$ time. Therefore, the overall time to initialize the priority queue is $\mathcal{O}(n \log n)$.

Initializing D_{CH} in line 7 takes $\mathcal{O}(n \log n)$ time [18]. In line 10, we can perform tangent queries on L^2 from t and v to find the first and last active points of u. In line 11, it will take no more than $\mathcal{O}(n)$ tangent queries to restore L^1 and L^2 throughout the algorithm by charging the queries to the points moved from the center convex hull to L^2 or from L^2 to L^1. Using an efficient dynamic convex hull data structure, it takes $\mathcal{O}(\log n)$ amortized time to delete a point and thus $\mathcal{O}(n \log n)$ time overall [18, 6]. We add points to the priority queue n times, delete points from the priority queue n times, and perform $\mathcal{O}(1)$ priority queue update operations for each iteration of the outer loop on line 8. Excluding lines 14 and 17 this establishes the overall $\mathcal{O}(n \log n)$ running time.

To bound the total time spent on line 14 to $\mathcal{O}(n \log n)$, we prove Theorem 5, bounding the total number of times points becomes active to $\mathcal{O}(n)$. Computing $\sigma(\bar{u})$ in line 14 requires us to find $A(\bar{u})$, where $\bar{u}$ is a new point added to the first layer. From the theorem, it takes $\mathcal{O}(n \log n)$ time to compute $A(\bar{u})$ for every $\bar{u}$. In addition, because it takes $\Theta(1 + |A(\bar{u})|)$ to compute $\sigma(\bar{u})$ from $A(\bar{u})$, overall it takes $\mathcal{O}(n)$ time to compute $\sigma(\bar{u})$ for every $\bar{u}$.

To bound the total time spent on line 17 on updating the sensitivities of u's neighbors to $\mathcal{O}(n \log n)$, we prove Lemma 6. Together with Theorem 5, it implies the desired result.

6 Geometric properties of peeling

In this section, we develop an amortized analysis of peeling to show that lines 14 and 17 can be computed efficiently. We ultimately aim to show that the number of times that any point becomes active for any triangle is $\mathcal{O}(n)$, bounding the amount of work done to initialize new triangles to $\mathcal{O}(n \log n)$. Then we show that the amount of work done to update the sensitivities of neighbor points is proportional to the number of new active points for them and an additive $\mathcal{O}(\log n)$ term. Thus, updating the sensitivities over all n iterations takes $\mathcal{O}(n \log n)$.

6.1 Preliminaries

When considering outer hull points, we use the notation $\triangle tuv$ for the triangle formed by u, its counterclockwise neighbor t, and its clockwise neighbor v. For a set of ordered vertices V we let $\circlearrowleft(V)$ be the polygon formed by the points in the (cyclical) order. We say $p \in \circlearrowleft(V)$ if p is strictly inside the polygon.

The following Lemma 2 combines a number of simple but useful propositions.

Lemma 2 *For a set of points P, the following propositions are true:*

1. *Any point $p \in P$ is active for at most three points on the first layer.*
2. *Let $\triangle tuv$ be a triangle for consecutive vertices (t, u, v) on the first layer and let $p \neq q$ be points $p \in \triangle tuv$ and $q \in \triangle tpv$. Then $q \notin A(u)$.*

3. *Let p be a point on any layer k. After deleting any point $q \neq p$ and reconstructing the convex layers, p is on layer $k-1$ or k.*

4. *Let (t, u, v) be consecutive vertices on the first layer L^1. Then if u is deleted, among the vertices in L^1, only the sensitivities of vertices t and v change.*

5. *For adjacent points (u, v) on the hull, $|A(u) \cap A(v)| \leq 1$.*

Proof. See Section 7.2 in the appendix. □

6.2 Bounding the active points

We will show that once a point is active for a hull point, it remains active for that hull point until the point is moved to the first layer. This implies a much stronger result by Lemma 2(1): over the entire course of the algorithm, a point becomes active for at most three other points. To do so, we first show that for each peel the active points $A(u)$ remain in u's triangle (Lemma 3) and second that the points in $A(u)$ remain active (Lemma 4).

Lemma 3 *Given a set of points P, for all adjacent hull points (u, v) and for all points $p \in A(u) \setminus A(v)$, if v is deleted then p still remains within u's triangle.*

Proof. Let t be u's other neighbor, and w.l.o.g. let the clockwise order on the hull be (t, u, v). Then if v' is u's new neighbor after deleting v, the clockwise order on the new hull will be (t, u, v'). Because p is active for u before v is deleted, $p \in \triangle tuv$.

First, we consider the case where $v' \notin \triangle tuv$. We want to show that $p \in \triangle tuv'$. Equivalently, that p is in the intersection of the three half-planes $\overrightarrow{tu}$, $\overrightarrow{uv'}$, and $\overrightarrow{tv'}$. Clearly, p must satisfy the half-planes $\overrightarrow{tu}$ and $\overrightarrow{uv'}$ as these coincide with hull edges. In addition, since $v' \notin \triangle tuv$, the half-plane for $\overrightarrow{tv}$ is a subset of the half-plane for $\overrightarrow{tv'}$. Because $p \in \triangle tuv$, p satisfies $\overrightarrow{tv}$. Therefore, p must satisfy $\overrightarrow{tv'}$.

Now we consider the case where $v' \in \triangle tuv$. Assume that $p \notin \triangle tuv'$. Then because we know that $p \in \triangle tuv$, either $p \in \triangle tv'v$ or $p \in \triangle uv'v$. If $p \in \triangle tv'v$, by Lemma 2(2), p could not have been active for u prior to deleting v. If $p \in \triangle uv'v$, p is now outside of the convex hull. Either way, this is a contradiction.

□

The following Lemma 4 shows that if p is in $A(u)$, it remains in $A(u)$ until moved to the first layer, after which it never becomes active again. It also shows that the active points $A(u)$ only change by adding or deleting points from either end, and thus can easily be found.

Lemma 4 *Given a set of points P, for all hull points u and v and for all points $p \in A(u) \setminus A(v)$, upon deleting v, p is in $A(u)'$, u's new set of active points.*

Proof.

Case 1 (u is not adjacent to v)

If u is not adjacent to v, there are no changes to $\triangle u$ upon deleting v, and thus, $A(u) = A(u)'$.

For the following cases, assume that u was adjacent to v. Then by Lemma 3, p is still in the triangle defined by u even after deleting v. Also, w.l.o.g. let (u, v) be the clockwise ordering of the points, and let v' be u's new neighbor.

Case 2 ($v' \in A(u)$)

By Lemma 2(5), $A(u) \cap A(v) = v'$. By Lemma 3, all points $A(u) \backslash \{v'\}$ are in $\triangle tuv'$. Because the second layer is a convex hull, each consecutive pair of points (a, b) in $t \circ A(u)$ define a half-plane $\overrightarrow{ab}$ with only points from the first layer to the left of each half-plane. This is still the case after deleting v by Lemma 3. Since the only new points on the first layer are $A(v)$ then all points in $A(u) \setminus \{v'\}$ remain on the second layer. Thus, the gift-wrapping starting from t wraps around all points in $A(u) \setminus \{v'\}$. Gift wrapping can hit no new points because, if that were true, there must be some point on the second layer to the left of one of the half-planes in described above. Thus, $A(u)' = A(u) \setminus \{v'\}$.

Case 3 ($v' \notin A(u)$)

Let u_e be the last point $A(u)$. Similar to the previous case, the gift-wrapping certifies all points in $A(u)$. Again, wrapping will not hit new active points before wrapping around u_e because that would imply the points hit were to the left of the half-planes described previously. When wrapping continues around u_e, several new active points may appear, until the wrapping terminates at v'. Thus, $A(u) \subseteq A(u)'$. □

Theorem 5 *For any 2D convex hull peeling process on n points the total number of times any point becomes active in any triangle is at most $3n$.*

Proof. This follows directly from the results of Lemma 2(1) and Lemma 4. □

6.3 Updating sensitivities

Next, we show that the total time to update the sensitivities in line 17 when peeling all n points takes $\mathcal{O}(\Delta + n \log n)$ time. Here Δ is the the number of times any point becomes active for any triangle. Theorem 5 proves that $\Delta = \mathcal{O}(n)$. The following lemma shows that the sensitivity of a point u can be updated in time proportional to the increase to $|A(u)|$ and an additive $\mathcal{O}(\log n)$ term. Figure 5 shows an example of how the sensitivity of a point changes when its neighbor is peeled.

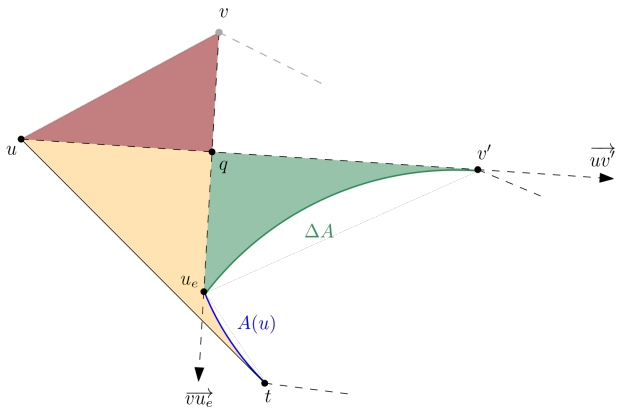


Figure 5: This figure shows how the sensitivity $\sigma(u)$ changes when point v is peeled. The point q is the intersection of the tangent from v to u_e and the tangent from u to v', where u_e is the last active point in $A(u)$ and v' is the first active point in $A(v)$. After the peel, v' replaces v as u's neighbor, and the points ΔA are newly active for u. The sensitivity $\sigma(u)$ before peeling v was equal to the area of $\pentagon(t \circ u \circ v \circ A(u))$. After peeling v, the sensitivity $\sigma(u)$ equals the area of $\pentagon(t \circ u \circ v' \circ \Delta A \circ A(u))$. Note how this can be computed in $\mathcal{O}(|\Delta A|)$ time from $\sigma(u)$ before the peel of v by subtracting the red area of $\triangle uvq$ and adding the green area of $\pentagon(u_e \circ q \circ v' \circ \Delta A)$.

Lemma 6 *Let (u, v) be points on the first layer. Consider a peel of v where δ_u new points become active points for u. Then the updated sensitivity $\sigma(u)$ can be computed it $\Theta(\delta_u + \log n)$ time, excluding the time to restore the second and first layer.*

Proof. The sensitivity $\sigma(u)$ is equal to the area of the polygon $U = \pentagon(t \circ u \circ v \circ A(u))$. By the *shoelace formula*, the area of U can be computed as the sum $S(U)$ of certain simple terms for each of its edges [5, 8]. We consider how U, and thus $S(U)$, changes when v is peeled. Inspecting the proof of Lemma 4, we see that at most two vertices are removed from U and at most $1 + \delta_u$ vertices are added to U. Furthermore, all the new vertices are located contiguously on the restored second layer and can be found in $\mathcal{O}(\delta_u + \log n)$ time using a tangent query from u's new neighbor which replaces v. To update $\sigma(u) = S(U)$, we simply add and subtract the appropriate $\mathcal{O}(\delta_u)$ terms depending on the removed and added edges.

□

7 Generalization and open problems

Theorem 7 shows that Algorithm 1 generalizes straightforwardly to other objectives such as peeling the point that causes the perimeter of the convex hull to decrease the most each iteration.

Theorem 7 *Let u be a point and O an objective where $\sigma_O(u)$ is the sensitivity of u under O. Consider the following three conditions:*

C1: *If $u \notin L^1$, then $\sigma_O(u) = 0$.*

C2: *If $u \in L^1$, then $\sigma_O(u) > 0$, and $\sigma_O(u)$ depends only on u, u's neighbors and its active points $A(u)$.*

C3: *If a single point p is added or removed from $A(u)$, then provided $\sigma_O(u)$ and the neighbors a_i and a_j of p in $A(u)$, the new sensitivity $\sigma_O(u)'$ can be computed in $\mathcal{O}(\log n)$ time.*

If O satisfies the above conditions, then Algorithm 1 runs in $\mathcal{O}(n \log n)$ time for objective O.

Proof. By conditions **C1** and **C2**, it is always a point u on the first layer that is peeled. Furthermore, when u is peeled only the sensitivities of the new points on the first layer and the neighbors of u must be updated since they are the only points for which their active points or neighbors change. Thus, Algorithm 1 can be used for objective O. Now we will show that the runtime of Algorithm 1 remains $\mathcal{O}(n \log n)$.

First, observe that all parts unrelated to computing sensitivities behave the same and still take $\mathcal{O}(n \log n)$ time. By condition **C3**, for a point u on the first layer, its sensitivity $\sigma_O(u)$ only depends on its neighbors and active points $A(u)$. As described in the proof of Lemma 6, when the set of points that affect $\sigma_O(u)$ changes, these points are readily available. The total number of neighbor changes is $\mathcal{O}(n)$ since, in each iteration, only the neighbors of the points adjacent to the peeled point change. The total number of changes to active points is $\mathcal{O}(n)$ by Theorem 5. If there are multiple changes to the active points in one iteration, such as when deleting one of u's neighbors, we perform one change at a time and, by condition **C3**, the total time to update sensitivities is $\mathcal{O}(n \log n)$. □

For concrete examples, we show how the three objectives *area* (O_A), *perimeter* (O_P), and *number of active points* (O_N) fit into this framework.

Let $f(\sigma(u), a_i, p, a_j) = \sigma(u) - d(a_i, a_j) + d(a_i, p) + d(p, a_j)$ be a function for computing the sensitivity $\sigma(u)$ when p is added to $A(u)$ between a_i and a_j (the functions where a point is removed from $A(u)$ or a neighbor of u changes are similar). For f to match each of the objectives it is sufficient to implement $d(\cdot, \cdot)$ as follows for points $a, b \in \mathbf{R}^2$:

O_A: $d(a, b) = \frac{1}{2}(a_2 b_1 - a_1 b_2)$

O_P: $d(a, b) = \sqrt{(b_2 - a_2)^2 + (b_1 - a_1)^2}$

O_N: $d(a, b) = 1$

The case with O_A is based on the shoelace formula. Additionally, for O_N to satisfy condition **C2**, we add 1 when computing the sensitivity of $u \in L^1$ to ensure that $\sigma(u) > 0$ even if $|A(u)| = 0$. For the three objectives, f takes $\mathcal{O}(1)$ time to compute satisfying the $\mathcal{O}(\log n)$ time requirement from condition **C3**.

7.1 Open problems

The first open problem is extending the result to $\mathbf{R}^3$ or higher. Directly applying our approach requires a dynamic 3D convex hull data structure, and Theorem 5 has to be extended to 3D. Second, is it possible to improve the quality of peeling by performing z-peels, even for $z = 2$ in $\mathcal{o}(n)$ time? Third, is there an efficient approximation algorithm for k-peeling?

Acknowledgement

We thank Asger Svenning for the initial discussions that inspired us to consider this problem.

References

[1] A. Aggarwal, H. Imai, N. Katoh, and S. Suri. Fining k points with minimum spanning trees and related problems. In *Proceedings of the fifth annual symposium on Computational geometry*, pages 283–291, 1989.

[2] G. Aloupis. Geometric measures of data depth. *DIMACS series in discrete mathematics and theoretical computer science*, 72:147, 2006.

[3] F. Angiulli and C. Pizzuti. Fast outlier detection in high dimensional spaces. In *European conference on principles of data mining and knowledge discovery*, pages 15–27. Springer, 2002.

[4] R. Atanassov, P. Bose, M. Couture, A. Maheshwari, P. Morin, M. Paquette, M. Smid, and S. Wuhrer. Algorithms for optimal outlier removal. *Journal of Discrete Algorithms*, 7(2):239–248, 2009. Selected papers from the 2nd Algorithms and Complexity in Durham Workshop ACiD 2006.

[5] R. Boland and J. Urrutia. Polygon area problems. In *Proc. of the 12th Canadian Conf. on Computational Geometry*, Fredericton, NB, Canada, 2000.

[6] G. Brodal and R. Jacob. Dynamic planar convex hull. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 617–626, 2002.

[7] B. Chazelle. On the convex layers of a planar set. *IEEE Transactions on Information Theory*, 31(4):509–517, 1985.

[8] F. Contreras. *Cutting polygons and a problem on illumination of stages.* University of Ottawa (Canada), 1998.

[9] R. N. Dave. Characterization and detection of noise in clustering. *Pattern Recognition Letters*, 12(11):657–664, 1991.

[10] M. De Berg. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000.

[11] D. P. Dobkin, R. Drysdale, and L. J. Guibas. Finding smallest polygons. *Computational Geometry*, 1:181–214, 1983.

[12] D. Eppstein. New algorithms for minimum area k-gons. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, page 83–88, USA, 1992. Society for Industrial and Applied Mathematics.

[13] D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete & Computational Geometry*, 11:321–350, 1994.

[14] D. Eppstein, M. Overmars, G. Rote, and G. Woeginger. Finding minimum area k-gons. *Discrete & Computational Geometry*, 7:45–58, 1992.

[15] F. E. Grubbs. *Sample criteria for testing outlying observations*. University of Michigan, 1949.

[16] S. Har-Peled. On the expected complexity of random convex hulls. *arXiv preprint arXiv:1111.5340*, 2011.

[17] A. Harsh, J. E., and P. Wei. Onion-peeling outlier detection in 2-d data sets. *International Journal of Computer Applications*, 139:26–31, 04 2016.

[18] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT Numerical Mathematics*, 32:249–267, 1992.

[19] P. J. Huber. The 1972 wald lecture robust statistics: A review. *The Annals of Mathematical Statistics*, 43(4):1041–1067, 1972.

[20] J. Hugg, E. Rafalin, K. Seyboth, and D. Souvaine. An experimental study of old and new depth measures. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 51–64. SIAM, 2006.

[21] D. Kirkpatrick and J. Snoeyink. Computing common tangents without a separating line. In S. G. Akl, F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, pages 183–193, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[22] E. M. Knorr and R. T. Ng. Finding intensional knowledge of distance-based outliers. In *Vldb*, volume 99, pages 211–222, 1999.

[23] S. K. Kwak and J. H. Kim. Statistical data preparation: management of missing values and outliers. *Korean journal of anesthesiology*, 70(4):407, 2017.

[24] M. Löffler and W. Mulzer. Unions of onions: preprocessing imprecise points for fast onion decomposition. *Journal of Computational Geometry*, 5(1), 2014.

[25] K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In *International Colloquium on Automata, Languages, and Programming*, pages 299–310. Springer, 2006.

[26] A. Meister. *Generalia de genesi figurarum planarum et inde pendentibus earum affectionibus*. 1769.

[27] M. H. Overmars and J. Van Leeuwen. Maintenance of configurations in the plane. *Journal of computer and System Sciences*, 23(2):166–204, 1981.

[28] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 427–438, 2000.

[29] M. Segal and K. Kedem. Enclosing k points in the smallest axis parallel rectangle. *Information Processing Letters*, 65(2):95–99, 1998.

[30] M. I. Shamos. *Problems in computational geometry*. 1975.

[31] A. F. Zuur, E. N. Ieno, and C. S. Elphick. A protocol for data exploration to avoid common statistical problems. *Methods in ecology and evolution*, 1(1):3–14, 2010.

Appendix

7.2 Proof of Lemma 2

Lemma 2(1) *Fix a point set P. Any point $p \in P$ is active in at most three triangles.*

Proof. First, note that a point can only be active for a hull point u if it is located inside $\triangle u$, so it is sufficient to show that any p is strictly inside at most three triangles. In addition, one can prove this by showing that $\triangle u$ only intersects with its neighbors' triangles $\triangle t$ and $\triangle v$.

Consider some $\triangle z$, such that z is not a neighbor of u. That is, u is not one of the vertices of $\triangle z$. If $\triangle z$ intersects with $\triangle u$, then either a vertex of $\triangle z$ is inside $\triangle u$ or the convex hull is a self-intersecting polygon, both violating convexity.

□

Lemma 2(2) *Let $\triangle tuv$ be a triangle for consecutive vertices (t, u, v) on the first layer and let $p \neq q$ be points $p \in \triangle tuv$ and $q \in \triangle tpv$. Then $q \notin A(u)$.*

Proof. By definition, $p \in \text{⌓}(t \circ v \circ A(u))$ or $p \in A(u)$. Either way, $q \in \triangle tpv$ implies that $q \in \text{⌓}(t \circ v \circ A(u))$, so $q \notin A(u)$. □

Lemma 2(3) *Let p be a point on any convex layer k. After deleting any point $q \neq p$ and reconstructing the convex layers, p is on layer $k-1$ or k.*

Proof. First we show that p never moves inward to layer $k' > k$. Consider the outermost layer L^1. By a property of convex hulls, every point v inside the convex hull is a convex combination of the hull points whereas any point $u \in L^1$ is not a convex combination of $L^1 - \{u\}$. If deleting q causes $p \in L^1$ to descend to a layer inside L^1, that implies that p is a convex combination of some subset of $P - \{q, p\}$. This contradicts the fact that p is not a convex combination of $L^1 - \{p\}$ and by extension is not a convex combination of $P - \{p\}$. Because of the recursive definition of convex layers, the proof for subsequent layers is symmetric.

Now we will show that p never moves up more than one layer at a time. This is clearly true for L^1 and L^2 because only one point is completely removed from the structure at at time (i.e. shifts to layer 0). For layers $k \geq 3$, consider a point p on layer k that moves to layer $k' \leq k-2$. Let $L^{*k'}$ be the set of points on layer k' after deleting q. Let L^{k-1} be the set of points on layer $k-1$ before deleting q.

Because $p \in L^{*k'}$, no convex combination of the points in $L^{*k'} - \{p\}$ equals p by convexity. By the inductive hypothesis, all points on L^{k-1} are convex combinations of $L^{*k'}$ because upon deleting q no point on L^{k-1} advances above layer k'. Furthermore, they are all convex combinations of $L^{*k'} - \{p\}$ as p itself is a convex combination of L^{k-1}. But if p is not a convex combination of $L^{*k'} - \{p\}$, and all the points on layer $k-1$ are convex combinations of $L^{*k'} - \{p\}$, then prior to deleting q, p was above layer $k-1$, which is a contradiction. □

Lemma 2(4) *Let (t, u, v) be consecutive vertices on the first layer L^1. Then if u is deleted, among the vertices in L^1, only the sensitivities of vertices t and v change.*

Proof. Consider a vertex z not adjacent to u. By the same arguments as in the proof of Lemma 2(1), the vertices defining $\triangle z$ do not change upon deleting u because it does not intersect $\triangle u$. In addition, because their triangles do not intersect, $|A(u) \cap A(z)| = 0$. Therefore, no points are removed from $A(z)$ upon deleting u.

Lastly, we will show that no points are added to $A(z)$ upon deleting u. Assume that there is some point p added to $A(z)$ when we delete u. But if p satisfies the conditions of being active for z and $\triangle z$ did not change upon deleting u, it should have been active for z before u was deleted, which is a contradiction.

Because $\triangle z$ and $A(z)$ do not change upon deleting u, it must be that $\sigma(z)$ remains the same. □

Lemma 2(5) *For adjacent points (u, v) on the hull, $|A(u) \cap A(v)| \leq 1$.*

Proof. We assume the contrary. Let $p \neq p'$ be two points such that $p, p' \in A(u) \cap A(v)$. By the definition of *active* and Lemma 2(3), p and p' must be on the second layer. W.l.o.g. let (u, v) be the clockwise ordering of the points on the first layer. In addition, let t be u's counterclockwise neighbor.

Say that p is the first point in $A(v)$. Then we have the tangent line $\overrightarrow{up}$ that defines p. By definition of tangent lines, no point on the second layer can be to the left of $\overrightarrow{up}$. But for p' to be active for v, then p' must be to the left of $\overrightarrow{pv}$. The only way to satisfy both half-planes is for p' to be placed such that $p \in \triangle tp'v$, in which case by Lemma 2(2) p cannot be in $A(u)$, which is a contradiction.

□

Chapter 11

The All Nearest Smaller Values Problem Revisited in Practice, Parallel and External Memory

The All Nearest Smaller Values Problem Revisited in Practice, Parallel and External Memory

Nodari Sitchinava
University of Hawaii at Mānoa
Honolulu, HI, USA
nodari@hawaii.edu

Rolf Svenning
Aarhus University
Aarhus, Denmark
rolfsvenning@cs.au.dk

ABSTRACT

We present a thorough investigation of the *All Nearest Smaller Values (ANSV)* problem from a practical perspective. The ANSV problem is defined as follows: given an array A consisting of n values, for each entry A_i compute the largest index $l < i$ and the smallest index $r > i$ such that $A_i > A_l$ and $A_i > A_r$, i.e., the indices of the nearest smaller values to the left and to the right of A_i. The ANSV problem was solved by Berkman, Schieber, and Vishkin [J. Algorithms, 1993] in the PRAM model. Their solution in the CREW PRAM model, which we will refer to as the *BSV* algorithm, achieves optimal $O(n)$ work and $O(\log n)$ span. Until now, the BSV algorithm has been perceived as too complicated for practical use, and we are not aware of any publicly available implementations. Instead, the best existing practical solution to the ANSV problem is the implementation by Shun and Zhao presented at DCC'13. They implemented a simpler $O(n \log n)$-work algorithm with an additional heuristic first proposed by Blelloch and Shun at ALENEX'11. We refer to this implementation as the *BSZ* algorithm. In this paper, we implement the original BSV algorithm and demonstrate its practical efficiency. Despite its perceived complexity, our results show that its performance is comparable to the BSZ algorithm. We also present the first theoretical analysis of the heuristic implemented in the BSZ algorithm and show that it provides a tunable trade-off between optimal work and optimal span. In particular, we show that it achieves $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ work and $O\left(k(1+\log \frac{n}{k})\right)$ span, for any integer parameter $1 \le k \le n$. Thus, for $k = \Theta(\log n)$, the BSZ algorithm can be made to be work-optimal, albeit at the expense of increased span compared to BSV. Our discussion includes a detailed examination of different input types, particularly highlighting that for random inputs, the low expected distance between values and their nearest smaller values renders simple algorithms efficient. Finally, we analyze the input/output (I/O) complexities of the BSV algorithm.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**.

Work supported by Independent Research Fund Denmark grant 9131-00113B and National Science Foundation grant CCF-1911245.

SPAA '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0416-1/24/06.
https://doi.org/10.1145/3626183.3659979

KEYWORDS

Algorithm analysis, parallel algorithms, external memory, PRAM, algorithm engineering, all nearest smaller values problem, ANSV

ACM Reference Format:
Nodari Sitchinava and Rolf Svenning. 2024. The All Nearest Smaller Values Problem Revisited in Practice, Parallel and External Memory. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3626183.3659979

1 INTRODUCTION

1.1 The All Nearest Smaller Values problem

In the All Nearest Smaller Values (ANSV) problem, we are given an array A consisting of n totally ordered elements $A_1, A_2, A_3, \ldots, A_n$, referred to as the *values*. The objective is to compute for each value the indices of the nearest smaller values to its left and right. Specifically, for a given value A_i, find the index l of the nearest smaller value A_l on its left. This index is $l = \max\{j \mid A_j < A_i \wedge j < i\}$, where A_l is termed the *left match* of A_i. Similarly, the nearest smaller value on the right, or the *right match*, should also be computed. The output of the problem is two arrays, L and R, each of size n. These arrays store the indices of each element's left and right matches in A, respectively. For simplicity, we extend A such that $A_0 = A_{n+1} = -\infty$, and let the index 0 indicate that a value has no left match, and the index $n+1$ that a value has no right match. All entries of L are initially 0, and all entries of R are initially $n+1$.

As is standard, we assume without loss of generality that all values in A are distinct. The case with equal values can be handled by simple modifications of the input and running the algorithm twice, once for left matches and once for right matches. We adopt the same notation as in [6] such that $l(A_i)$ and $r(A_i)$ denote the indices of the left and right match of A_i, respectively.

ANSV is a fundamental problem since many problems directly reduce to an ANSV computation or ANSV can be used as an important subroutine. A non-exhaustive list of such problems is: finding the min/max among n elements, merging sorted lists, constructing Cartesian trees [23], monotone polygon triangulation, range minimum queries, parenthesis matching, binary tree reconstruction. See [4–6] for more details.

To highlight one example, consider merging two sorted lists, $a = a_1, a_2, a_3, \ldots, a_n$ and $b = b_1, b_2, b_3, \ldots, b_m$. This task directly reduces to computing the ANSV on $a \circ \text{rev}(b)$, where $\text{rev}(b)$ denotes the reverse of list b, and $\circ$ represents concatenation. Specifically, the position of each a_i and b_j in the merged list can be determined as $n + m - r(a_i) + i + 1$ and $l(b_j) + j + 1$, respectively, where $r(a_i)$ and $l(b_j)$ are the indices of the right and left matches for a_i and b_j in $a \circ \text{rev}(b)$.

1.2 Models of computation

For analyzing algorithms, we focus on the CREW PRAM [14, 15] and the external memory (EM) [1] models of computation and the only allowed operations on values are comparisons. The CREW PRAM model is characterized by multiple processors operating synchronously on shared memory with concurrent reads (CR) and exclusive writes (EW) capabilities. Algorithms in this model are analyzed in terms of *work* – the total number of operations performed by all processors, and *span* – the depth of the longest computation path using an infinite number of processors. We adopt the notation T_P to denote the time it takes to execute an algorithm on a P-processor CREW PRAM. Then work and span correspond to T_1 and T_∞, respectively. Analyzing just work and span is sufficient because Brent's scheduling principle [9] can then be used to obtain the runtime T_P for an arbitrary number of processors $P \geq 1$ as $T_P = O\left(\frac{T_1}{P} + T_\infty\right)$.

The EM model by Aggarwal and Vitter [1] (also known as the *ideal-cache model* [11][1]) is characterized by a processor with an internal memory of size M and an infinite external memory. Initially, the input of size n is placed in $\lceil \frac{n}{B} \rceil$ consecutive blocks of B consecutive elements in the external memory. Each processor can perform *input/output* (I/O) operations to move a block of elements between the external and internal memories and data must be in the internal memory to perform any computation on it. The cost in this model is the number of I/Os performed. If an algorithm does not use the parameters M and B in its description, it is referred to as being *cache-oblivious* [11].

Arge et al. [2] extended the EM model to the parallel setting. The *Parallel External Memory (PEM)* [2] model consists of P processors, each containing internal memory of size M and sharing the external memory. The I/Os are performed in parallel, with a *parallel I/O* consisting of up to P processors transferring one block in each time step. Since no equivalent to Brent's scheduling principle exists in the PEM model, analysis of parallel I/Os must be performed for a specific value of P.

1.3 Previous results

Sequentially, the ANSV problem can be solved in linear time and $O\left(\frac{n}{B}\right)$ I/Os cache-obliviously using a stack to push elements from left to right. Before each element A_i is pushed on the stack, pop all elements larger than A_i from the stack. The remaining element at the top of the stack is smaller than A_i and is the left match of A_i. Right matches can be found similarly. This algorithm behaves exactly like the stack-based algorithm in [12] for constructing Cartesian trees. Instead of a stack, a simple implementation using arrays can also be employed [3].

We call the stack/array-based algorithm *SEQ*. These approaches are based on the following basic observation.

[1]The main difference between the EM and the ideal-cache models is that in the ideal-cache model the transfers between the internal and external memories are delegated to a separate omniscient paging algorithm. Therefore, the paging algorithm can optimize the I/Os based on its knowledge of future accesses and performs no worse than any explicitly stated block transfer algorithm in the EM model. The well-known resource-augmentation result [11] states that any reasonable automated paging algorithm, e.g., the one that evicts only the least-recently-used (LRU) block from the internal memory, with internal memory size of $2M$ performs asymptotically similarly to the omniscient paging algorithm with internal memory of size of M.

Observation 1. *The matches in the ANSV problem are non-overlapping. That is, for any value A_i with a right match r_i, there is no other value A_j for $j < i$ with a right match r_j, such that $i < r_j < r_i$ (likewise for the left match).*

In the parallel setting, Berkman, Schieber, and Vishkin (BSV) [6] presented an optimal $O(n)$ work and $O(\log \log n)$ span algorithm in CRCW PRAM and an optimal $O(n)$ work and $O(\log n)$ span algorithm in CREW PRAM. The latter is our focus and we call this the *BSV* algorithm. In the literature, their work-efficient algorithm is perceived as being *"very complicated"* [8, 19]. They also presented a much simpler $O(n \log n)$ work and $O(\log n)$ span algorithm which we call the *work-inefficient BSV* algorithm. The algorithm proceeds in two stages, first, it constructs in $O(n)$ work and $O(\log n)$ span (see [6] section 3.2) a balanced binary tree with the values A stored in its leaves in the original order. Each internal node then takes the value of the minimum of its children. We call this a *min binary tree*. Second, to find the left match of A_i it follows the path towards the root until a left child has a value smaller than A_i. From that child, it follows a path towards the leaves always choosing the right child if its value is smaller than A_i. It is straightforward to see that this finds the left match $l(A_i)$ in $O(\log n)$ time and symmetrically the same for right matches. This algorithm was implemented for parallel Lempel-Ziv factorization by Shun and Zhao [20] and Cartesian tree construction by Blelloch and Shun [8, 19] where ANSV was used as a subroutine. Interestingly, Blelloch and Shun added a surprisingly effective heuristic, and in [8] they write:

> *"... we note that the ANSV only takes about 10% of the total time even though it is an $O(n \log n)$ algorithm. This is likely due to the optimization discussed above."*

In their paper, the role of the heuristic was not emphasized as a critical element, and its operational details were somewhat unclear to us.

However, this heuristic is implemented in the publicly available code [21], and we provide an in-depth explanation of it in Section 1.4. In this paper, we analyze the heuristic and show that it results in a provably better work than $O(n \log n)$ and provides a tunable trade-off between work and span.

Generally, the BSV algorithm generalizes well to parallel models other than PRAM and has been adapted in various other models. For example, it has been used to solve ANSV in the bulk synchronous parallel model [13] and a formal derivation using Coq [18]. It has been implemented in the Distributed Memory models, both in theory [16] and in practice using MPI [10]. In the hypercube model, the ANSV was solved in optimal $O(\log n)$ time with n processors [17].

1.4 The BSZ algorithm

In this section, we describe the BSZ algorithm and the heuristic in detail. We begin with the heuristic which is based on an integer parameter $1 \leq k \leq n$ and modifies the simple $O(n \log n)$ work-inefficient algorithm [6] in three ways. :

- **H1** Partition the input into $\lceil \frac{n}{k} \rceil$ blocks of size k (except the last block which may not be full). For each block run the sequential ANSV algorithm to find all matches within the block, we call these *local* matches.
- **H2** When using the min-binary tree to search for a match, perform an exponential search in the direction of the match,

to find it in $O(\log d)$ time, where d is the distance in the input to the match. To do this, in addition to following parent pointers also move horizontally in the tree. In the code, they facilitate this by implementing the binary tree as an array of arrays, where the secondary arrays store the nodes at a given height.

H3 For each block, the elements without a right match form an increasing sequence (Observations 1a and 1b in [6]). Instead of finding the remaining right matches in parallel, do so sequentially from right to left. When performing an exponential search for the right match, start from where the previous search ended. Symmetrically for the left matches.

In short and focusing on the right matches, computing the local matches in each block leaves an increasing sequence of $1 \le d \le k$ unmatched values at indices $u_1, u_2, u_3, ..., u_d$. Thus, proceeding from right to left, for each pair of adjacent unmatched values $A_{u_{i-1}}$ and A_{u_i}, the search in the min binary tree for the match of $A_{u_{i-1}}$ can start from where A_{u_i} found its match.

The BSZ algorithm works on a global array A of n values and stores the right matches in an array R of n matches, all initialized to $n+1$ (we omit the left matches for simplicity). It is supplied with the hyperparameter k and has access to a min binary tree T on A, which can be built in $O(n)$ work and $O(\log n)$ span. We adopt the Python notation for subarrays where $A[x : y]$ denotes $[A_i \mid x \le i < y]$ and the notation $A^i = A[(i-1)k + 1 : \min(ik\,,\ n) + 1]$ for the ith block of A. The BSZ algorithm is described in pseudocode as Algorithm 1.

Algorithm 1: The BSZ algorithm for ANSV

```
Output: Computes the right matches of A and stores them
        in R
1 for i = 1 to ⌈n/k⌉ in parallel do
2     Compute local matches in A^i using the SEQ algorithm
        and store them in R^i
3     start ⟵ min(ik, n) + 1
4     for j = min(ik, n) down to (i − 1)k + 1 do
5         if R_j == n + 1 then
6             R_j ⟵ matchRight(start, j) // traverses T
                from index start for the match of A_j
7             start ⟵ R_j
```

1.5 The BSV algorithm

This section presents an overview of the BSV algorithm as described in [6]. The algorithm is described in pseudocode as Algorithm 2. Like the BSZ algorithm, the BSV algorithm operates on global arrays A and R, each of size n, representing values and their right matches, respectively. For simplicity, we omit left matches. The algorithm uses a min binary tree T based on A and is supplied with a hyperparameter k. Initially, the algorithm computes the local matches within each block A^i. It also determines the index i_m of the smallest value A_{i_m} in each block, along with its left and right matches $l(A_{i_m})$ and $r(A_{i_m})$, respectively. This latter is done by traversing the min binary tree T. The results are stored in array M. While using M is not mandatory, it prevents redundant searches in the tree. Then, for each block, the algorithm identifies the boundaries of a merging problem in constant time based on the contents of M (refer to Lemmas 3.3 and 3.4 in [6] for details). For $1 \le a < b < c < d \le n$ the two subsequences $A[a : b]$ and $A[c : d]$ that are merged may be far apart. Through this merging process, the algorithm identifies all the right matches for $A[a : b]$ and the left matches for $A[c : d]$. Note that each block defines at most two merging problems, leading to a total of $O\left(\frac{n}{k}\right)$ merging problems. By setting $k = \Theta(\log n)$, traversing the tree a constant number of times for each block results in $\Theta(n)$ work. Similarly, computing local matches within a group or performing a merge can be accomplished in $\Theta(k) = \Theta(\log n)$ time.

Algorithm 2: The BSV algorithm for ANSV

```
Output: Computes the right matches of A and stores them
        in R
1 M ⟵ Array of size ⌈n/k⌉ // For storing information
    to identify merging problems
2 for i = 1 to ⌈n/k⌉ in parallel do
3     Compute local matches in A^i using the SEQ algorithm
        and store them in R^i
4     i_m ⟵ index of min value in A^i
5     M[i] ⟵ {i_m, l(A_{i_m}), r(A_{i_m})} // Uses T
6 for i = 1 to ⌈n/k⌉ in parallel do
7     (a, b, c, d) ⟵ mergeBoundaries(i) // Computes
        boundaries of a merging problem. Uses M
        instead of searching in the tree T
8     merge(a, b, c, d) // Computes nonlocal matches by
        merging A[a : b] and A[c : d]
```

2 OUR RESULTS

We give the first theoretical analysis of the heuristic used in the BSZ algorithm which improves on the simple $O(n \log n)$ work and $O(\log n)$ span algorithm in [6]. We show that it provides a tunable trade-off between optimal work and optimal span for the hyperparameter $1 \le k \le n$. In particular, we show that it achieves $O\left(n\left(1 + \frac{\log n}{k}\right)\right)$ work and $O\left(k(1 + \log \frac{n}{k})\right)$ span, for any integer $1 \le k \le n$. Note that setting $k = 1$ corresponds to the work-inefficient BSV algorithm; setting $k = \Theta(\log n)$ achieves linear work, matching the work of the BSV algorithm, but resulting in the $O\left(\log n \log \frac{n}{k}\right)$ span; and setting $k = n$ corresponds to the SEQ algorithm.

Second, we present the first implementation of the BSV algorithm for shared memory machines, to our knowledge. Although the BSV algorithm has been perceived as being theoretically complicated, our implementation is a simple ~175 line C++ implementation, with ~90 lines reused directly from the publicly available ~120 line implementation of the BSZ algorithm from [20]. Our implementation is comparable with the current state-of-the-art BSZ implementation and achieves parallel speedup of up to 13.7 on 24 cores (48 threads with hyper-threading). We also verify experimentally that the heuristic introduced in the BSZ algorithm significantly speeds up the algorithm and reduces the work to $O(n)$ for large enough k.

Third, we show that when each value is drawn i.i.d. from a discrete distribution on a totally ordered set of size m the expected distance from a value to its match is at most H_m – the mth Harmonic number. Thus, these random inputs are not hard instances for this problem, as even the trivial $O(n^2)$ solution for ANSV is expected to achieve $O(n \log n)$ work and $O(\log n)$ span if $m = O(poly(n))$. Similarly, the work-inefficient BSV algorithm achieves $O(n \log \log n)$ work and $O(\log \log n)$ span.

Finally, we present the first I/O complexity analysis of the BSV algorithm in the (P)EM model. As with many other parallel models, the BSV algorithm generalizes well in the PEM model too and we show that simple modifications to the BSV algorithm yield $O(\frac{n}{PB} + \log_B n)$ parallel I/Os for any positive integer $P \geq 1$ of processors. Finally, we show that the array-based version like the stack-based version of the SEQ algorithm uses $O(n/B)$ I/Os cache-obliviously.

3 ANALYZING THE BSZ ALGORITHM

In this section, we analyze the heuristic introduced in the BSZ algorithm and show that it provides a tunable trade-off between work and span as described by the following Theorem.

Theorem 1. *For an input of size n and any integer hyperparameter $1 \leq k \leq n$, the BSZ algorithm achieves $O\left(n\left(1 + \frac{\log n}{k}\right)\right)$ work and $O\left(k(1 + \log \frac{n}{k})\right)$ span.*

We use the terminology that when a value finds its match in the same block as itself we call it a *local* match. Likewise, when a value finds its match in a different block than itself we call it a *nonlocal* match. We begin by analyzing the span.

Lemma 1. *For an input of size n and any integer hyperparameter $1 \leq k \leq n$, the span of the BSZ algorithm is $O\left(k(1 + \log \frac{n}{k})\right)$.*

Proof of Lemma 1. The algorithm has three parts. First, constructing the min binary tree takes $O(\log n)$ time. Second, finding the local matches in a block takes $O(k)$ time. Third, using the heuristic to sequentially find the nonlocal left matches in a block takes $O\left(\sum_{i=1}^{k}(1 + \log n_i)\right)$ time where n_i denotes the distance between the $(i-1)$th and ith match and the plus one accounts for any case $n_i \leq 1$.[2] The sum is then upper bounded as follows: $\sum_{i=1}^{k}(1 + \log n_i) \leq k + \sum_{i=1}^{k} \log \frac{n-k}{k} \leq k\left(1 + \log \frac{n}{k}\right)$. Adding all parts together gives $O\left(k\left(1 + \log \frac{n}{k}\right) + \log n\right) = O\left(k(1 + \log \frac{n}{k})\right)$ span. □

Next, we will prove the following lemma, which bounds the work W_{BSZ} of the BSZ algorithm:

Lemma 2. *Let W_{BSZ} be the work of the BSZ algorithm on an input of size n using the hyperparameter k. Then $W_{BSZ} = O\left(n\left(1 + \frac{\log n}{k}\right)\right)$.*

To prove Lemma 2, we will focus on a slightly different recursive algorithm, which we call *REC*, for the ANSV problem. We stress that this algorithm is only used for the analysis of the work of the BSZ algorithm. The idea is that this algorithm is simpler to analyze and uses about the same work as the BSZ algorithm. Like the BSZ algorithm, the REC algorithm operates on global arrays A and R of size n, with R being initialized to $n+1$, storing the values and right matches, respectively (left matches are omitted for simplicity). It also uses a min binary tree T on A, which can be built using $O(n)$ work, and is supplied a hyperparameter k. The REC algorithm is described in pseudocode as Algorithm 3, and its behavior on a specific input is exemplified in Figure 1.

Algorithm 3: The REC(x, y) algorithm for ANSV

Input: Indices $x \leq y$
Output: Computes the right matches of values $A[x : y]$ and stores them in $R[x : y]$

```
1  if x == y then return ;
2  x_k ⟵ min(x + k , y)
3  Compute local matches in A[x : x_k] using the SEQ
     algorithm and store them in R[x : x_k]
4  start ⟵ x_k
5  for i = x_k − 1 down to x do
6      if R_i == n + 1 then
7          R_i ⟵ matchRight(start, i) // traverses T from
             index start for the match R_i of A_i
8          REC(start, R_i)
9          start ⟵ R_i
10 REC(start, y)
```

To solve the ANSV problem the initial call is REC$(1, n+1)$. The REC algorithm, guided by the heuristics, identifies disjoint parts of A that can be solved independently. To do so, for the first block of k values, it runs the SEQ algorithm. Among these, some will find their match locally within the block. It uses the min binary tree T from right to left for the remaining matches using the same heuristics (**H2** and **H3**) as the BSZ algorithm. The indices of these nonlocal matches partition the remaining $n - k$ values into at most $k + 1$ disjoint subproblems which can be solved independently. The following Lemma makes that precise.

Lemma 3. *For any call to REC(x, y), the right matches of all values in $A[x : y]$ are in $A[x : y + 1]$.*

Proof. Follows directly from Observation 1 that all matches are non-overlapping. □

To bound the work of the BSZ algorithm using the REC algorithm, we will first prove Lemma 4, which established that they use about the same work.

Lemma 4. *Let W_{BSZ} and W_{REC} be the work of the BSZ and REC algorithms for a particular input of size n using the same value for the hyperparameter k. Then $W_{BSZ} = O\left(W_{REC} + n\left(1 + \frac{\log n}{k}\right)\right)$.*

The converse (swapping W_{BSZ} and W_{REC}) also holds, but this direction is not required for our analysis.

Next, we will bound the work of the REC algorithm. For clarity of exposition, let $W_{REC} = O(T(n))$, i.e., denote an upper bound on the work of the recursive algorithm on an input of size n, excluding

[2] We adopt the convention that $\log 0 = 0$.

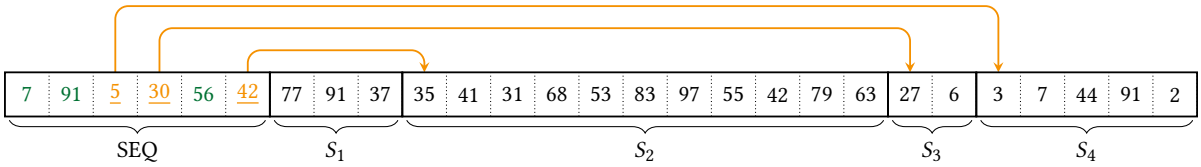


Figure 1: The REC algorithm begins by running the SEQ algorithm on the first $k = 6$ values (line 3), finding local matches for 7, 91, and 56. The other three values, 5, 30, and 42, have nonlocal matches, indicated by arrows, which are found using the min binary tree T. These values partition the remaining input into four independent subproblems: S_1, S_2, S_3, and S_4. For instance, the recursive call for S_1 is REC$(7, 10)$ (line 8). For the last subproblem S_4, the recursive call is REC$(23, 28)$ (line 10).

the construction of the min binary tree. Then, $T(n)$ is defined by the following recursion:

$$T(n) = \begin{cases} k + \max\left(\sum_{i=1}^{k+1} T(n_i) + \sum_{i=1}^{k} \log(n_i)\right) & n > k \\ n & n \le k, \end{cases}$$

where the maximum is defined over all possible partitions of the input into $k+1$ subproblems, each of size n_i, such that $\sum_{i=1}^{k+1} n_i = n - k$ (because the k values that define the partitions are already matched), and the sum of logarithmic terms comes from performing the non-local searches in the min tree using heuristic **H2**. Lemma 5 bounds $T(n)$ and demonstrates that $T(n)$ decreases as the block size k increases:

Lemma 5. $T(n) = O\left(n\left(1 + \frac{\log n}{k}\right)\right)$.

Together, Lemmas 4 and 5 will imply the bound on the work of the BSZ algorithm as stated in Lemma 2.

We first prove Lemma 4, which shows that the work of the BSZ and REC algorithms is about the same. We fix an input A of size n and focus solely on the right matches, as the behavior of the left matches is symmetric. The analysis begins by splitting the work of the BSZ and REC algorithms into two parts. The first part is constructing the binary tree and computing local matches in each block. The second part is finding the nonlocal match of each element. Thus, $W_{BSZ} = \Theta\left(n + \sum_{i=1}^{n} z_i\right)$ and $W_{REC} = \Theta\left(n + \sum_{i=1}^{n} r_i\right)$, where z_i and r_i denote the number of nodes visited in the min binary tree when finding the ith nonlocal right match of value A_i by the BSZ and REC algorithms, respectively. The count is 0 if the match is found locally or the search starts from the index of the match, and it is at least 1 otherwise. At a high level, our strategy will be to bound the difference between $\sum_{i=1}^{n} z_i$ and $\sum_{i=1}^{n} r_i$ for each block of size k. More precisely, for the BSZ algorithm, consider blocks of indices $Z = \left[Z^1, Z^2, Z^3, ..., Z^{\lceil n/k \rceil}\right]$, each corresponding to indices where local matches are computed on line 2. That is, $Z^i = [j \mid (i-1)k + 1 \le j < \min(ik, n) + 1]$, noting that all blocks are of size k, except possibly the last one. For the REC algorithm, consider $m < n$ blocks of indices $R = \left[R^1, R^2, R^3, ..., R^m\right]$, where each R^i corresponds to indices where local matches are computed on line 3. Specifically, if a recursive call REC(x, y) computes local matches in $A[x : x_k]$ on line 3, it yields a block of indices $[j \mid x \le j < x_k]$. Each block R^i has a maximum size of k.

Proof of Lemma 4. We begin by showing that $\left|\sum_{j \in Z^i} z_j - r_j\right| = O(k + \log n)$ for any $1 \le i \le \lceil n/k \rceil$. Consider running the BSZ algorithm on a fixed input resulting in Z^i. Also consider running the REC algorithm on the same input and focus on the subset of $c \le k$ blocks $\left[R^{i_1}, R^{i_2}, R^{i_3}, ..., R^{i_c}\right] = \left[R^\ell \mid Z^i \cap R^\ell \ne \emptyset \wedge 1 \le \ell \le m\right]$ that overlap with Z^i.

Case 1 ($c = 1$).

Since a block from the REC algorithm perfectly overlaps with Z^i, exactly the same nodes in the min binary tree (with repetition) are visited, resulting in $\sum_{j \in Z^i} z_j - r_j = 0$.

Case 2. *[$c = 2$]*

Here the blocks R^{i_1} and R^{i_2} split Z^i into two parts. Consequently, they split the $1 \le d \le k$ unmatched values at indices $u_1, u_2, u_3, ..., u_d$ for the BSV algorithm in Z^i. Let $\bar{s}$ be the last index of R^{i_1} where the split occurs. Consider the most general case when the split is strictly between two unmatched values at indices $u_s < \bar{s} < u_{s+1}$ for $1 \le s \le d - 1$. See Figure 2 for an example. The cases where the split occurs before $s < u_1$, after $u_d < s$, or overlaps with some u_* are simpler.

In the part of Z^i to the left of the split, i.e., in $Z^i \cap R^{i_1}$, the nonlocal matches for the BSZ algorithm are also nonlocal for the REC algorithm. Now, there are only two places where the running time between the two algorithms may differ. First, since r_{u_s} does not start its search in the min binary tree from where $r_{u_{s+1}}$ found its match, it follows that $0 \le r_{u_s} - z_{u_s} = O(\log n)$. Second, all values in $[u_s + 1, \bar{s}]$ are matched locally for the BSZ algorithm but some of them may be unmatched for the REC algorithm. We denote these unmatched values by R_+ and establish that $0 \le \sum_{j \in R_+} r_j - z_j = \sum_{j \in R_+} r_j = O\left(|R_+| \log \frac{k}{|R_+|}\right) = O(k)$, using the concavity of the logarithm and that $\sum_{j \in R_+} r_j$ actually corresponds to $\Theta(\sum_i \log n_i)$, where $\sum_i n_i \le k$.

Next, we consider the part of Z^i after the split, i.e., $Z^i \cap R^{i_2}$. If r_{u_d} is not the last nonlocal match in R^{i_2}, then it is the only place where a different number of nodes of the min binary tree are visited, and the difference is $0 \le z_{u_d} - r_{u_d} = O(\log n)$. If r_{u_d} is the last nonlocal match in R^{i_2}, then exactly the same nodes are visited. Finally, there may be multiple unmatched values for the BSZ algorithm that are matched locally for the REC algorithm starting with r_{u_d}. We denote these local matches by R_-. As previously, we establish that $0 \le \sum_{j \in R_-} z_j - r_j = \sum_{j \in R_-} z_j = O\left(|R_-| \log \frac{k}{|R_-|}\right) = O(k)$. For the last nonlocal match at d_r, if it exists, the difference is $0 \le r_{d_r} - z_{d_r} = O(\log n)$. Combining all the contributions results in $\left|\sum_{j \in Z^i} z_j - r_j\right| = O(k + \log n)$, concluding this case.

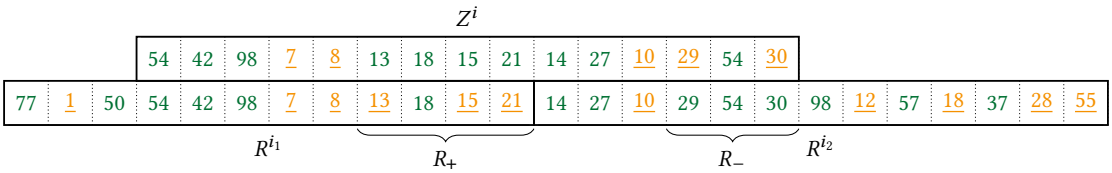


Figure 2: Case 2 in the proof of Lemma 4, where exactly two blocks, R^{i_1} and R^{i_2}, overlap with the block Z^i. The numbers that are underlined are unmatched in their respective blocks. For example, the $d = 5$ unmatched values in Z^i are 7, 8, 10, 29, and 30, and the split $\bar{s}$ occurs strictly between the unmatched values 8 and 10 at indices $u_2 = u_s$ and $u_3 = u_{s+1}$, respectively. The values that are unmatched in R^{i_1} but matched in Z^i, are 13, 15, and 21, corresponding to R_+. The values that are matched in R^{i_2} but unmatched in Z^i, are 29 and 30, corresponding to R_-. The braces indicate the range where values in R_+ or R_- are located.

Case 3 ($3 \le c$).

Consider any block R^{i_t} for $2 \le t \le c-1$ and the corresponding recursive call REC(x, y) with $y \le n$, which computed local matches at R^{i_t}. Since $R^{i_t} \subsetneq Z^i$ then $\left|R^{i_t}\right| = y - x < k$, and all nonlocal matches in R^{i_t} match A_y, by Lemma 3. For the REC algorithm, since the search starts from index y (line 4), no nodes are visited in the min binary tree. For the BSZ algorithm, since $y \in Z^i$, then all the values at R^{i_t} are matched locally, and no nodes of min binary tree are visited. Thus, the running times may differ only at R^{i_1} and R^{i_c}, which is similar to the case for $c = 2$.

We have now established that $\left|\sum_{j \in Z^i} z_j - r_j\right| = O(k + \log n)$ for any $1 \le i \le \lceil n/k \rceil$. Summing over each block Z^i concludes the proof:

$$\begin{aligned} W_{BSZ} &= \Theta\left(n + \sum_{i=1}^{n} z_i\right) \\ &= \Theta(n) + O\left(\sum_{i=1}^{\lceil n/k \rceil} \sum_{j \in Z^i} r_j + k + \log n\right) \\ &= \Theta(n) + O\left(\left(\sum_{i=1}^{n} r_i\right) + \lceil n/k \rceil \, (k + \log n)\right) \\ &= O\left(W_{REC} + n\left(1 + \frac{\log n}{k}\right)\right) \end{aligned}$$

□

Next, we prove Lemma 5, which shows that the work to find nonlocal matches decreases as k increases.

Proof of Lemma 5. We will prove that $T(n) \le 2n + \left(\frac{n}{k} - 1\right) \log n$ by induction. In the base case, when $n \le k$, $T(n) = n \le 2n + \left(\frac{n}{k} - 1\right) \log n$. For the inductive case:

$$\begin{aligned} T(n) &= k + \max\left(\sum_{i=1}^{k+1} T(n_i) + \sum_{i=1}^{k} \log n_i\right) \\ &\le k + \max\left(\sum_{i=1}^{k+1} \left(2n_i + \left(\frac{n_i}{k} - 1\right) \log n_i\right) + \sum_{i=1}^{k} \log n_i\right) \\ &\le k + 2(n-k) + \frac{1}{k} \cdot \max\left(\sum_{i=1}^{k+1} n_i \log n_i\right) \\ &\le 2n - k + \frac{1}{k}\left(\sum_{i=1}^{k+1} n_i\right) \log \sum_{i=1}^{k+1} n_i \\ &< 2n + \left(\frac{n}{k} - 1\right) \log n \end{aligned}$$

□

Using Lemmas 4 and 5, it is now straightforward to bound the work of the BSZ algorithm, which concludes the proofs of Lemma 2 and Theorem 3.

4 RANDOM INPUTS

Consider a random input where each input value is drawn independently and identically distributed from a discrete distribution over a totally ordered set of size m. Then, the expected distance between a value and its match (here the first smaller or equal value) is strictly less than $\sum_{i=1}^{m} p_i \frac{1}{\sum_{j=1}^{i} p_j}$. The strictness follows since the array is bounded. For example, with a uniform distribution, the expected distance is strictly less than H_m. Thus, for $m = O(poly(n))$ the expected distance is $\Theta(\log n)$. The arguably simplest ANSV algorithm is a double for-loop that scans left and right for the match of each value in parallel. For the uniform distribution with m as discussed earlier, this algorithm achieves expected $O(n \log n)$ work. Similarly, the work-inefficient algorithm with heuristic **H2** spends $O(\log d)$ time on finding the match of a value that is at a distance of d from its match. Thus, using Jensen's Inequality, it achieves expected $O(n \log \log n)$ work. For this reason, we consider random inputs to be easy.

5 EXTERNAL MEMORY

In this section, we prove Theorems 2 and 3, which give the I/O complexity of the BSV (Algorithm 2) and array-based SEQ algorithms in the PEM model.

Theorem 2. *For any $P \geq 1$, the ANSV problem can be solved on the P-processor PEM model on an input of size n in $\Theta\left(\frac{n}{PB} + \log_B n\right)$ parallel I/Os.*

Proof. Set group sizes to $k = \Theta(B \log_B n)$ and replace the binary tree with a B-tree. Then run the same BSV algorithm in $\lceil \frac{n}{kP} \rceil$ rounds, each round processing a contiguous segment of kP elements.

The straightforward bottom-up parallel construction of the B-tree takes $O\left(\frac{n}{PB} + \log_B n\right)$ parallel I/Os. In each round, merging is done sequentially by each processor, with each processor spending $O\left(\frac{k}{B}\right) = O(\log_B n)$ I/Os per round. Finally, in each round, each processor traverses the B-tree once, resulting in $\Theta(\log_B n)$ parallel I/Os per round. Combining the I/Os over the $\lceil \frac{n}{kP} \rceil$ rounds results in overall $O\left(\frac{n}{PB} + \log_B n\right)$ parallel I/O complexity. □

Theorem 3. *The array-based SEQ algorithm is cache-oblivious, and for an input of size n, it uses $O\left(\frac{n}{B}\right)$ I/Os.*

Proof. The SEQ algorithm sequentially finds the left (similarly right) matches by looping over A from left to right. It maintains the invariant that in the ith iteration, all left matches of values $A[1:i]$ have been found and are stored in the array $L[1:i]$. To find the left match of A_i, the algorithm simply follows the matches previously found in L, starting with L_{i-1}, until it finds a value $A_j < A_i$, and then sets $L[i] = j$. Given that the matches are non-overlapping (as noted in Observation 1), this result is not too surprising.

The amortized analysis of the I/O complexity is the same as for the number of comparisons in the RAM model [12], but with a potential of one I/O for each block instead of the individual elements. In particular, in the ith iteration, define the potential to be the number of blocks currently hit by the path generated by following the pointers starting from $L[i-1]$. Without going through all the cases, when an insertion (that is not the first in a block and itself extends the path) causes $3 \leq k$ blocks to be visited on the path, then the new path will hit $k-2$ fewer blocks, and the potential can pay for the visited blocks. □

6 EXPERIMENTS

In this section, we investigate the performance of the BSZ and BSV algorithms in practice. We show that even though the BSV algorithm has been perceived as theoretically complicated, the code is simple, and it achieves comparable performance to the current state-of-the-art implementation. Our code is available online [22]. We also confirm experimentally that the heuristic introduced for the BSZ algorithm is effective, and it significantly speeds up the algorithm and reduces the work to be linear for a large enough k.

6.1 Experimental setup

The experiments were run on two Intel Xeon Silver 4214 2.20GHz 12-core CPUs distributed across 2 sockets with hyper-threading enabled, totaling 48 threads and 126GB of shared RAM. The cache configuration included a 32K L1 cache, a 1024K L2 cache, and a 16896K L3 cache. Our implementation is in C++ 17 and compiled using GCC 7.5.0 with the `-O3` optimization. All inputs are of type `long` (8 bytes). For parallelization, we used the ParlayLib library [7], which supports parallel loops with `parlay::parallel_for` and `parlay::blocked_for`. For consistency with the BSZ implementation [8, 20], we use basic arrays instead of `parlay::sequences` and switched their parallel loops from using Cilk to ParlayLib. Our simple C++ implementation of the BSV algorithm uses ~175 lines of code, of which ~90 are reused directly from the BSZ implementation, which totals ~120 lines. Both algorithms use a min binary tree and find local matches using the SEQ algorithm. They differ in their approach to finding nonlocal matches. The BSV algorithm uses merging, while the BSZ algorithm searches within the tree. We simplified the merging in the BSV algorithm by ignoring already matched elements. This contrasts with the original description in [6] steps 6.1 and 6.2, which uses prefix and suffix minimas to identify the unmatched values. For the SEQ algorithm, we decided to use the array-based implementation since we found it to be about 30% faster than the stack-based implementation.

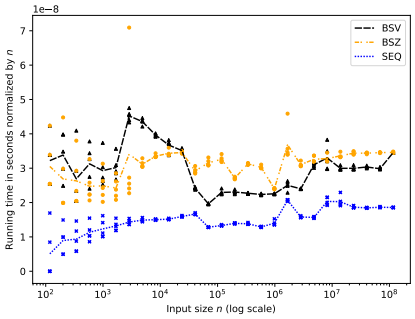
	P=1			P=48	
	SEQ	BSZ	BSV	BSZ	BSV
Sorted	1.03	1.95	2.26	0.27	0.17
Random	3.25	4.86	7.39	0.27	0.31
Merge	1.27	2.51	2.33	0.34	0.24
RandomMerge	2.16	4.03	4.02	0.34	0.24

Table 1: Running times in seconds for the SEQ, BSZ and BSV algorithms on the Sorted, Random, Merge and RandomMerge inputs for $P = 1$ and $P = 48$. We report the average of 5 runs, each with $n = \lfloor 1.7^{35} \rfloor = 116335496$ and block size $k = 256\lfloor \log_2 n \rfloor = 6656$.

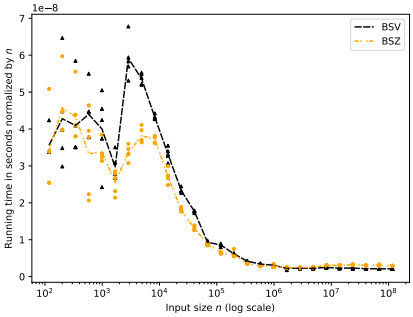
We considered 4 different types of inputs. First, Sorted: the of numbers $1, 2, 3, \ldots, n$. Second, Random: a random permutation of $1, 2, 3, \ldots, n$. Third, Merge: the numbers $0, 2, 4, \ldots, \lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 3, \ldots, 1$, corresponding to the reduction from merging sorted lists to ANSV where the two sorted lists must be perfectly interleaved (for example, 0, 2, 4 and 1, 3, 5 forming input 0, 2, 4, 5, 3, 1). Fourth, RandomMerge: similar to the Merge input, except the two values in each consecutive pair are swapped with probability 0.5.

6.2 Performance

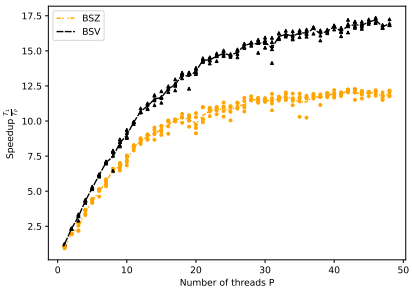
In Table 1, we list the average running time in seconds for the three different algorithms across the four types of inputs, both for $P = 1$ and $P = 48$ threads, with $n \approx 10^8$ and block size $k = 256\lfloor \log_2 n \rfloor = 6656$. The block size is chosen to achieve $\Theta(n)$ work, and the constant 256 was determined through initial experiments. In section 6.3, we explore the impact of the block size k in more depth. The Sorted input is a trivial ANSV instance and was primarily used as a baseline to gauge how the algorithms should perform on an easy input. In practice, it also turned out to be the fastest. The Random input was the slowest overall for $P = 1$, whereas for $P = 48$, there was no decidedly slowest input type. We suspected the slowdown was due to additional branch mispredictions, which we investigated using the *perf* command. The results in Table 2



(a) Running time in seconds normalized by n for the SEQ, BSZ, and BSV algorithms with $P = 1$.



(b) Running time in seconds normalized by n for the BSZ and BSV algorithms with $P = 48$.



(c) The speedup T_1/T_P for the SEQ, BSZ, and BSV algorithms for $P = 1, 2, 3, \ldots, 48$.

Figure 3: For all three plots, each dot corresponds to the running time in seconds of an algorithm on the RandomMerge input. For each input size n, we repeat the experiment 5 times and draw a line through the average of these 5 runs. In each run we set the block size $k = 256 \log n$ to ensure $\Theta(n)$ work. For plots 3a and 3b we use a log scale and test inputs of size $n = 1.7^p$ for $p = 1, 2, 3, \ldots$ and $n \leq 2^{27} = 134,217,728$. The running time is in seconds normalized by n (see the 1e-8 on the axis).

	P=1		P=48	
	BSZ	BSV	BSZ	BSV
Sorted	1.6	1.6	5.0	4.8
Random	258.4	425.5	267.0	437.2
Merge	1.4	1.4	4.1	4.1
RandomMerge	138.5	129.4	144.8	137.4

Table 2: Branch mispredictions in millions for the BSZ and BSV algorithms for the same parameters as in Table 1.

	T_1/T_{12}	T_1/T_{24}	T_1/T_{48}
BSZ	8.59	10.86	11.98
BSV	8.64	11.99	13.71

Table 3: Speedup of algorithms BSZ and BSV for 12, 24, and 48 processors. Speedup is calculated as T_1/T_p, where T_1 is the running time for 1 processor and T_p is the running time for p processors.

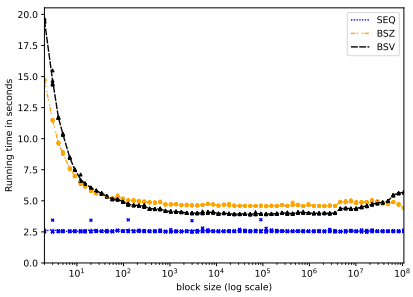
provide evidence of this for $P = 1$. Based on the discussion in section 4, we decided not to focus on the random inputs in further experiments. The behavior of the Merge input is comparable to that of RandomMerge, experiencing only a 20-30% slowdown. We believe the latter is the most well-motivated for three reasons. First, it naturally occurs in the reduction from merging sorted lists to ANSV. Second, without the heuristic, the BSZ algorithm performs $\Theta(n \log n)$ work. Third, there are many far-away matches which are the hard ones to compute. Taking inspiration from the Random input, we added some randomness, giving us the RandomMerge input, which, as expected for $P = 1$, is $60 - 80\%$ slower. Across the four inputs, for $P = 48$, the BSV algorithm is comparable to or slightly faster than the BSZ algorithm.

In Figure 3, we plot increasing n against running time in seconds normalized by n for each algorithm on the Merge input. In plot 3a where $P = 1$, we observe a mostly flat trend as expected, since all algorithms perform $\Theta(n)$ work. Even though there is a slight trend upward, the variance and performance of the BSZ and BSV algorithms mimic the SEQ algorithm, which serves as a simple baseline for what $\Theta(n)$ work should look like. In plot 3b where $P = 48$, both the BSZ and BSV algorithms converge nicely at around $0.26 \cdot 10^{-8}$ seconds, with a running time of ~ 0.305 seconds normalized by $n = 116335496$.

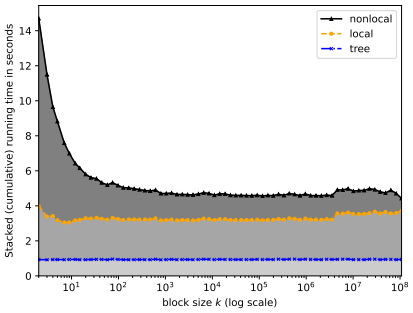
The speedup of a parallel algorithm is the ratio between its sequential running time T_1 with $P = 1$ processors and its running time T_P with P processors. In Figure 3 plot 3c, we plot the speedup of the BSZ and BSV algorithms as we increase the number of threads $P = 1, 2, 3, \ldots, 48$. The final speedup T_1/T_{48} is 11.98 and 13.71 for the BSZ and BSV algorithms, respectively. Until around $P = 12$ we observe a strong linear speedup, likely because one of the two CPUs with 12 cores is active and no hyper-threading is activated yet. From 12 to 24 processors, the speedup continues to increase steadily for both algorithms. From about 24 processors onwards, the speedup tapers off, but still increases more for the BSV algorithm than for the BSZ algorithm. See Table 3 for the exact speedups for $P = 12$ and $P = 24$.

6.3 Block size

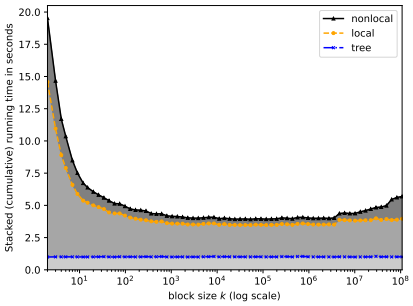
Both the BSZ and BSV algorithms use a hyperparameter k for the block size, where in each block local matches are found sequentially. The primary distinction between the algorithms lies in their approach for handling the remaining nonlocal matches. This section explores the impact of the block size k on the different components of the algorithms. We categorize the running time into three parts. First, the *tree* part denotes the time to construct the min binary tree for both algorithms. Second, the *local* part represents the time to compute local matches in both algorithms, and for the BSV algorithm, it also includes the time to set up the merging problems. Third, the *nonlocal* part denotes the time spent traversing the min binary tree to find nonlocal matches in the BSZ algorithm. For the BSV algorithm it denotes the time spent on finding nonlocal



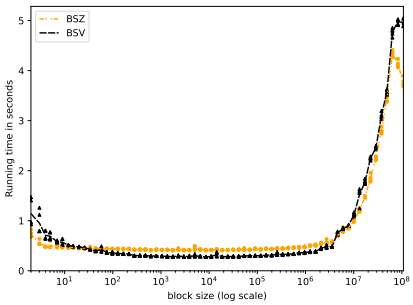
(a) Running time in seconds for increasing block size k with $P = 1$ for the SEQ, BSZ, and BSV algorithms.



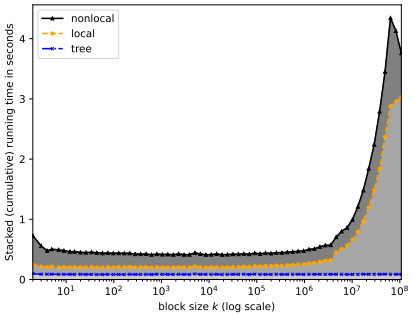
(b) The stacked running time in seconds for increasing block size k with $P = 1$ for the BSZ algorithm.



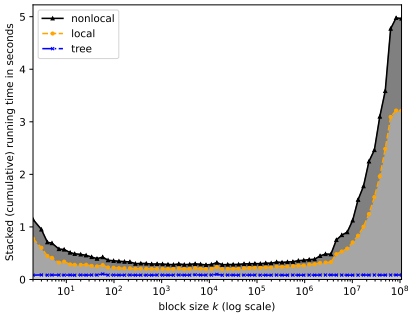
(c) The stacked running time in seconds for increasing block size k with $P = 1$ for the BSV algorithm.



(d) Running time in seconds for increasing block size k with $P = 48$.



(e) The stacked running time in seconds for increasing block size k with $P = 48$ for the BSZ algorithm.



(f) The stacked running time in seconds for increasing block size k with $P = 48$ for the BSV algorithm.

Figure 4: For all six plots, we measured the average running time for increasing block size k (log scale) on the RandomMerge input of fixed size $n = 2^{27} = 134,217,728$, repeated 5 times, and drew a line through those averages. For the three top plots 4a, 4b and 4c we have $P = 1$, and for the three bottom plots 4d, 4e and 4f we have $P = 48$. In plots 4c, 4c, 4e and 4f, we show the stacked running time in seconds for the three parts of the BSZ and BSV algorithms. The *tree* part denotes the time to construct the min binary tree for both algorithms. The *local* part denotes the time to compute local matches for both algorithms, and for the BSV algorithm also the time to set up the merging problems. The *nonlocal* part denotes for the BSZ algorithm time spent traversing the min binary tree for nonlocal matches. For the BSV algorithm, it denotes the time spent on finding nonlocal matches by merging.

matches by merging. All parts take $\Theta(n)$ work, except for the nonlocal part of the BSZ algorithm, which takes $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ time, as given by Lemma 2. Similarly, for the BSV algorithm, the local part takes $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ time, due to the necessity of traversing the min binary tree twice for each block to set up the merging problems. For the RandomMerge input, the parts depending on k are $O\left(n\left(1+\frac{\log n}{k}\right)\right)$ as expected. Figure 4 plot 4a clearly shows this behavior, with the work decreasing rapidly as k increases. Plots 4b and 4c further confirm that it is nonlocal and local parts for the BSZ and BSV algorithms, respectively, that decrease, and that the others parts are independent of k. For $P = 48$, we still see an improvement in running time for small k in Figure 4 plot 4d. Not surprisingly, for large k, the running time increases dramatically, as both algorithms solve each block sequentially.

7 ACKNOWLEDGMENTS

The authors express gratitude to the Data-Intensive Systems group at Aarhus University for their provision of computing resources, and to the anonymous reviewers whose constructive feedback contributed to the improved presentation of the paper.

REFERENCES

[1] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM* 31, 9 (1988), 1116–1127. https://doi.org/10.1145/48529.48535

[2] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. 2008. Fundamental Parallel Algorithms for Private-Cache Chip Multiprocessors. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*. 197–206. https://doi.org/10.1145/1378533.1378573

[3] Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. 2012. LRM-Trees: Compressed Indices, Adaptive Sorting, and Compressed Permutations. *Theoretical Computer Science* 459 (2012), 26–41. https://doi.org/10.1016/j.tcs.2012.08.010

[4] O. Berkman, Dany Breslauer, Zvi Galil, Baruch Schieber, and Uzi Vishkin. 1989. Highly Parallelizable Problems. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing* (Seattle, Washington, USA)

(STOC '89). Association for Computing Machinery, New York, NY, USA, 309–319. https://doi.org/10.1145/73007.73036

[5] O Berkman, B Schieber, and U Vishkin. 1988. Some Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. In *Technical Report UMIACS-TR-88-79*. Univ. of Maryland Inst. for Advanced Computer Studies New York/Berlin.

[6] O. Berkman, B. Schieber, and U. Vishkin. 1993. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. *Journal of Algorithms* 14, 3 (1993), 344–370. https://doi.org/10.1006/jagm.1993.1018

[7] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib-A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20)*. 507–509. https://doi.org/10.1145/3350755.3400254

[8] Guy E. Blelloch and Julian Shun. 2011. A Simple Parallel Cartesian Tree Algorithm and Its Application to Suffix Tree Construction. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. 48–58. https://doi.org/10.1137/1.9781611972917.5

[9] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (1974), 201–206. https://doi.org/10.1145/321812.321815

[10] Patrick Flick and Srinivas Aluru. 2017. Parallel Construction of Suffix Trees and the All-Nearest-Smaller-Values Problem. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 12–21. https://doi.org/10.1109/IPDPS.2017.62

[11] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 285–297. https://doi.org/10.1109/SFFCS.1999.814600

[12] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. 1984. Scaling and Related Techniques for Geometry Problems. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (STOC '84)*. Association for Computing Machinery, New York, NY, USA, 135–143. https://doi.org/10.1145/800057.808675

[13] Xin He and Chun-Hsi Huang. 2001. Communication Efficient BSP Algorithm for All Nearest Smaller Values Problem. *J. Parallel and Distrib. Comput.* 61, 10 (2001), 1425–1438. https://doi.org/10.1006/jpdc.2001.1741

[14] Joseph JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

[15] Richard M. Karp and Vijaya Ramachandran. 1991. *Parallel Algorithms for Shared-Memory Machines*. MIT Press, Cambridge, MA, USA, 869–941.

[16] Jyrki Katajainen. 1996. Finding All Nearest Smaller Values on a Distributed Memory Machine. In *Proceedings of the Computing: The 2nd Australasian Theory Symposium (Australian Computer Science Communications, Vol. 18)*. Computer Science Association (Australia), 100–107.

[17] D. Kravets and C.G. Plaxton. 1994. An Optimal Hypercube Algorithm for the All Nearest Smaller Values Problem. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*. 505–512. https://doi.org/10.1109/SPDP.1994.346129

[18] Frédéric Loulergue, Simon Robillard, Julien Tesson, Joeffrey Legaux, and Zhenjiang Hu. 2014. Formal Derivation and Extraction of a Parallel Program for the All Nearest Smaller Values Problem. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (Gyeongju, Republic of Korea) *(SAC '14)*. Association for Computing Machinery, New York, NY, USA, 1577–1584. https://doi.org/10.1145/2554850.2554912

[19] Julian Shun and Guy E. Blelloch. 2014. A Simple Parallel Cartesian Tree Algorithm and Its Application to Parallel Suffix Tree Construction. *ACM Trans. Parallel Comput.* 1, 1, Article 8 (10 2014), 20 pages. https://doi.org/10.1145/2661653

[20] Julian Shun and Fuyao Zhao. 2013. Practical Parallel Lempel-Ziv Factorization. In *2013 Data Compression Conference*. IEEE, 123–132. https://doi.org/10.1109/DCC.2013.20

[21] Julian Shun and Fuyao Zhao. 2013. Practical Parallel Lempel-Ziv Factorization. https://github.com/zfy0701/Parallel-LZ77/blob/release. Accessed: 2023-10-01.

[22] Nodari Sitchinava and Rolf Svenning. 2024. A Parallel Implementation of an Optimal ANSV Algorithm. https://github.com/algoparc/ANSV/.

[23] Jean Vuillemin. 1980. A Unifying Look at Data Structures. *Commun. ACM* 23, 4 (1980), 229–239. https://doi.org/10.1145/358841.358852

Bibliography

[1] Mikkel Abrahamsen and Bartosz Walczak. Common tangents of two disjoint polygons in linear time and constant workspace. *ACM Trans. Algorithms*, 15 (1), December 2018. ISSN 1549-6325. doi: 10.1145/3284355. 44

[2] Mikkel Abrahamsen, Anna Adamaszek, and Tillmann Miltzow. The art gallery problem is exist r-complete. *J. ACM*, 69(1), December 2021. ISSN 0004-5411. doi: 10.1145/3486220. URL https://doi.org/10.1145/3486220. 43

[3] Mikkel Abrahamsen, Joakim Blikstad, André Nusser, and Hanwen Zhang. Minimum star partitions of simple polygons in polynomial time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC 2024, page 904–910, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703836. doi: 10.1145/3618260.3649756. URL https://doi.org/10.1145/3618260.3649756. 43

[4] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962. In Russian. English translation in Soviet Mathematics - Doklady, 3:1259–1263, 1962. 18

[5] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988. ISSN 0001-0782. doi: 10.1145/48529.48535. URL https://doi.org/10.1145/48529.48535. 10

[6] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. ISSN 1432-0541. doi: 10.1007/s00453-003-1021-x. URL https://doi.org/10.1007/s00453-003-1021-x. 12, 18

[7] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003. doi: 10.1137/S009753970240481X. 25, 26

[8] Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to

graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007. doi: 10.1137/S0097539703428324. 11

[9] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, page 197–206, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939739. doi: 10.1145/1378533.1378573. URL https://doi.org/10.1145/1378533.1378573. 31

[10] Lars Arge, Morten Revsbaek, and Norbert Zeh. I/o-efficient computation of water flow across a terrain. In *Proceedings of the Twenty-Sixth Annual Symposium on Computational Geometry*, SoCG '10, page 403–412, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300162. doi: 10.1145/1810959.1811026. URL https://doi.org/10.1145/1810959.1811026. 10

[11] Lars Arge, Gerth Stølting Brodal, and S. Srinivasa Rao. External memory planar point location with logarithmic updates. *Algorithmica*, 63(1):457–475, 2012. ISSN 1432-0541. doi: 10.1007/s00453-011-9541-2. 25

[12] Lars Arge, Mathias Rav, Sarfraz Raza, and Morten Revsbæk. *I/O-Efficient Event Based Depression Flood Risk*, pages 259–269. Society for Industrial and Applied Mathematics, 2017. doi: 10.1137/1.9781611974768.21. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611974768.21. 10

[13] Rossen Atanassov, Prosenjit Bose, Mathieu Couture, Anil Maheshwari, Pat Morin, Michel Paquette, Michiel Smid, and Stefanie Wuhrer. Algorithms for optimal outlier removal. *Journal of Discrete Algorithms*, 7(2):239–248, 2009. ISSN 1570-8667. doi: https://doi.org/10.1016/j.jda.2008.12.002. Selected papers from the 2nd Algorithms and Complexity in Durham Workshop ACiD 2006. 38, 41

[14] Avis and Toussaint. An optimal algorithm for determining the visibility of a polygon from an edge. *IEEE Transactions on Computers*, C-30(12):910–914, 1981. doi: 10.1109/TC.1981.1675729. 44

[15] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359579. URL https://doi.org/10.1145/359576.359579. 27

[16] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery. ISBN

9781450379410. doi: 10.1145/1734663.1734671. URL https://doi.org/10.1145/1734663.1734671. 7, 9, 18

[17] Djamal Belazzougui, Gerth Stølting Brodal, and Jesper Sindahl Nielsen. Expected linear time sorting for word size ω(log2n loglogn). In R. Ravi and Inge Li Gørtz, editors, *Algorithm Theory – SWAT 2014*, pages 26–37, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08404-6. 10

[18] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to B-trees and write-optimization. *;login:*, 40(5), 2015. URL https://www.usenix.org/publications/login/oct15/bender. 19

[19] Michael A. Bender, Rathish Das, Martín Farach-Colton, Rob Johnson, and William Kuszmaul. Flushing without cascades. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 650–669, USA, 2020. Society for Industrial and Applied Mathematics. 18

[20] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2020.07.063. URL https://www.sciencedirect.com/science/article/pii/S0377221720306895. 5

[21] Jon L. Bentley. Solutions to klee's rectangle problems. Unpublished manuscript, Department of Computer Science, Carnegie Mellon University, 1977. 25

[22] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1993.1018. URL https://www.sciencedirect.com/science/article/pii/S0196677483710187. 7, 32, 34

[23] Ahmad Biniaz, Anil Maheshwari, Magnus Christian Ring Merrild, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, Eliot W. Robson, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, Thomas Shermer, Jack Spalding-Jamieson, Rolf Svenning, and Da Wei Zheng. Polynomial-Time Algorithms for Contiguous Art Gallery and Related Problems. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry (SoCG 2025)*, volume 332 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:21, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-370-6. doi: 10.4230/LIPIcs.SoCG.2025.20. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SoCG.2025.20. 6, 42

[24] Ahmad Biniaz, Anil Maheshwari, Joseph S. B. Mitchell, Saeed Odak, Valentin Polishchuk, and Thomas Shermer. Contiguous boundary guarding, 2025. URL https://arxiv.org/abs/2412.15053. 7, 42

[25] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990. ISBN 026202313X. 31

[26] Guy E. Blelloch and Julian Shun. *A Simple Parallel Cartesian Tree Algorithm and its Application to Suffix Tree Construction*, pages 48–58. Society for Industrial and Applied Mathematics, 2011. doi: 10.1137/1.9781611972917.5. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611972917.5. 32, 34

[27] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. ISSN 0022-0000. doi: https://doi.org/10.1016/S0022-0000(73)80033-9. URL https://www.sciencedirect.com/science/article/pii/S0022000073800339. 14, 16

[28] Klaus Brengel, Andreas Crauser, Paolo Ferragina, and Ulrich Meyer. An experimental study of priority queues in external memory. *ACM J. Exp. Algorithmics*, 5:17–es, December 2001. ISSN 1084-6654. doi: 10.1145/351827.384259. URL https://doi.org/10.1145/351827.384259. 12

[29] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974. ISSN 0004-5411. doi: 10.1145/321812.321815. URL https://doi.org/10.1145/321812.321815. 31

[30] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, page 52–58, USA, 1996. Society for Industrial and Applied Mathematics. ISBN 0898713668. 11, 12

[31] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, page 307–315, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136749. doi: 10.1145/780542.780589. URL https://doi.org/10.1145/780542.780589. 11

[32] Gerth Stølting Brodal and Jyrki Katajainen. Worst-case external-memory priority queues. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, SWAT '98, page 107–118, Berlin, Heidelberg, 1998. Springer-Verlag. ISBN 3540646825. 12

[33] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 1177–1184, New York, NY, USA, 2012.

Association for Computing Machinery. ISBN 9781450312455. doi: 10.1145/2213977.2214082. URL https://doi.org/10.1145/2213977.2214082. 12

[34] Gerth Stølting Brodal, Casper Moldrup Rysgaard, Jens Kristian Refsgaard Schou, and Rolf Svenning. Space-efficient functional offline-partially-persistent trees with applications to planar point location. In Pat Morin and Subhash Suri, editors, *Algorithms and Data Structures*, pages 644–659, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-38906-1. 6

[35] Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. External memory fully persistent search trees. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, page 1410–1423, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399135. doi: 10.1145/3564246.3585140. URL https://doi.org/10.1145/3564246.3585140. 6, 25

[36] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. *ACM Trans. Algorithms*, 21(2), January 2025. ISSN 1549-6325. doi: 10.1145/3707692. URL https://doi.org/10.1145/3707692. 12

[37] Gerth Stølting Brodal, Spyros Sioutas, Konstantinos Tsakalidis, and Kostas Tsichlas. Fully persistent b-trees. *Theoretical Computer Science*, 841: 10–26, 2020. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2020.06.027. URL https://www.sciencedirect.com/science/article/pii/S0304397520303686. 23

[38] Gerth Stølting Brodal, Michael T. Goodrich, John Iacono, Jared Lo, Ulrich Meyer, Victor Pagan, Nodari Sitchinava, and Rolf Svenning. External-memory priority queues with optimal insertions. In *Proceedings of the 33rd Annual European Symposium on Algorithms (ESA 2025)*, 2025. URL https://tildeweb.au.dk/au121/papers/esa25pq.pdf. To appear at ESA 2025. 5, 11, 12, 13, 16

[39] Gerth Stølting Brodal, Casper Moldrup Rysgaard, and Rolf Svenning. Buffered partially-persistent external-memory search trees. In *Proceedings of the 33rd Annual European Symposium on Algorithms (ESA 2025)*, 2025. URL https://arxiv.org/abs/2503.08211. To appear in ESA 2025; Preprint available at arXiv:2503.08211. 6

[40] G.S. Brodal and R. Jacob. Dynamic planar convex hull. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 617–626, 2002. doi: 10.1109/SFCS.2002.1181985. 39

[41] John Canny. Some algebraic and geometric computations in pspace. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*,

STOC '88, page 460–467, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912640. doi: 10.1145/62212.62257. URL https://doi.org/10.1145/62212.62257. 42

[42] James A Carlson, Arthur Jaffe, and Andrew Wiles. *The millennium prize problems*. American Mathematical Soc., 2006. 5

[43] B. Chazelle. On the convex layers of a planar set. *IEEE Transactions on Information Theory*, 31(4):509–517, 1985. doi: 10.1109/TIT.1985.1057060. 38, 40

[44] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1):133–162, 1986. ISSN 1432-0541. doi: 10.1007/BF01840440. 25

[45] Edmund M. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69659-9. 5

[46] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. ISBN 9781450374644. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047. 5, 42

[47] William K. Cornwell, Dylan W. Schwilk, and David D. Ackerly. A trait-based test for habitat filtering: Convex hull volume. *Ecology*, 87(6):1465–1471, 2006. doi: https://doi.org/10.1890/0012-9658(2006)87[1465:ATTFHF]2.0.CO;2. 41

[48] V.A. Crupi, S.K. Das, and M.C. Pinotti. Parallel and distributed meldable priority queues based on binomial heaps. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, volume 1, pages 255–262 vol.1, 1996. doi: 10.1109/ICPP.1996.537167. 11

[49] Andrew Danner, Thomas Mølhave, Ke Yi, Pankaj K. Agarwal, Lars Arge, and Helena Mitasova. Terrastream: from elevation data to watershed hierarchies. In *Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems*, GIS '07, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595939142. doi: 10.1145/1341012.1341049. URL https://doi.org/10.1145/1341012.1341049. 10

[50] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691. 5, 10

[51] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf. 5, 10

[52] Rathish Das, John Iacono, and Yakov Nekrich. External-Memory Dictionaries with Worst-Case Update Cost. In Sang Won Bae and Heejin Park, editors, *33rd International Symposium on Algorithms and Computation (ISAAC 2022)*, volume 248 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:13, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-258-7. doi: 10.4230/LIPIcs.ISAAC.2022.21. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ISAAC.2022.21. 18

[53] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. *ACM Trans. Algorithms*, 3(2):13–es, May 2007. ISSN 1549-6325. doi: 10.1145/1240233.1240236. URL https://doi.org/10.1145/1240233.1240236. 23

[54] Peter J. Denning. The locality principle. *Communications of the ACM*, 48 (7):19–24, 07 2005. ISSN 0001-0782. doi: 10.1145/1070838.1070856. URL https://cir.nii.ac.jp/crid/1364233268784653952. 3

[55] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 365–372, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912217. doi: 10.1145/28395.28434. URL https://doi.org/10.1145/28395.28434. 17, 21, 23, 26

[56] Paul F. Dietz and Rajeev Raman. Persistence, randomization and parallelization: On some combinatorial games and their applications (abstract). In Frank Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures*, pages 289–301, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-47918-5. 17, 21

[57] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959. ISSN 0029-599X. doi: 10.1007/BF01386390. URL https://doi.org/10.1007/BF01386390. 11

[58] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1): 86–124, 1989. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(89)90034-2. URL https://www.sciencedirect.com/science/article/pii/0022000089900342. 4, 17, 19, 21, 26, 27, 28

[59] H El Gindy and D Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2(2):186–197, 1981. ISSN 0196-6774. doi: https://doi.org/10.1016/0196-6774(81)90019-5. 44

[60] David Eppstein. New algorithms for minimum area k-gons. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, page 83–88, USA, 1992. Society for Industrial and Applied Mathematics. ISBN 089791466X. URL https://dl.acm.org/doi/abs/10.5555/139404.139422. 38

[61] David Eppstein. Decremental greedy polygons and polyhedra without sharp angles. arXiv:2507.04538, 2025. To appear in Proc. CCCG 2025. 41

[62] David Eppstein, Mark Overmars, Günter Rote, and Gerhard Woeginger. Finding minimum area k-gons. *Discrete & Computational Geometry*, 7:45–58, 1992. doi: 10.1007/BF02187823. 38

[63] R. Fadel, K.V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(99)00006-7. URL https://www.sciencedirect.com/science/article/pii/S0304397599000067. 12, 13, 14

[64] Arash Farzan, Alejandro López-Ortiz, Patrick K. Nicholson, and Alejandro Salinger. Algorithms in the ultra-wide word model. In Rahul Jain, Sanjay Jain, and Frank Stephan, editors, *Theory and Applications of Models of Computation*, pages 335–346, Cham, 2015. Springer International Publishing. ISBN 978-3-319-17142-5. 31

[65] Robert W Floyd. Algorithm 245: treesort. *Communications of the ACM*, 7(12):701, 1964. 11

[66] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 114–118, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450374378. doi: 10.1145/800133.804339. URL https://doi.org/10.1145/800133.804339. 10, 31

[67] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 285–297, 1999. doi: 10.1109/SFFCS.1999.814600. 11

[68] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1), January 2012. ISSN 1549-6325. doi: 10.1145/2071379.2071383. URL https://doi.org/10.1145/2071379.2071383. 11

[69] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984. ISSN 0025-5661. doi: 10.1007/BF01744431. URL https://doi.org/10.1007/BF01744431. 10

[70] R Garey Michael and S Johnson David. Computers and intractability: A guide to the theory of np-completeness, 1979. 5

[71] Wayne M. Getz and Christopher C. Wilmers. A local nearest-neighbor convex-hull construction of home ranges and utilization distributions. *Ecography*, 27 (4):489–505, 2004. doi: https://doi.org/10.1111/j.0906-7590.2004.03835.x. 41

[72] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995. ISSN 0004-5411. doi: 10.1145/227683.227684. URL https://doi.org/10.1145/227683.227684. 5

[73] Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 89–94, New York, NY, USA, 1978. Association for Computing Machinery. ISBN 9781450374378. doi: 10.1145/800133.804336. URL https://doi.org/10.1145/800133.804336. 10

[74] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, 1978. doi: 10.1109/SFCS.1978.3. 18

[75] S. L. Hakimi. Optimum locations of switching centers and the absolute centers and medians of a graph. *Operations Research*, 12(3):450–459, 1964. doi: 10.1287/opre.12.3.450. 41

[76] Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/o-efficient generation of massive graphs following the lfr benchmark. *ACM J. Exp. Algorithmics*, 23, August 2018. ISSN 1084-6654. doi: 10.1145/3230743. URL https://doi.org/10.1145/3230743. 10

[77] John Hershberger and Subhash Suri. Applications of a semi-dynamic convex hull algorithm. *BIT Numerical Mathematics*, 32:249–267, 1992. doi: 10.1007/BF01994880. 39

[78] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986. ISSN 0001-0782. doi: 10.1145/7902.7903. URL https://doi.org/10.1145/7902.7903. 32

[79] Peter J Huber. The 1972 wald lecture robust statistics: A review. *The Annals of Mathematical Statistics*, 43(4):1041–1067, 1972. 37, 41

[80] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17(2):157–184, 1982. ISSN 1432-0525. doi: 10.1007/BF00288968. URL https://doi.org/10.1007/BF00288968. 18

[81] J. Hughes. Why functional programming matters. *The Computer Journal*, 32 (2):98–107, 01 1989. ISSN 0010-4620. doi: 10.1093/comjnl/32.2.98. URL https://doi.org/10.1093/comjnl/32.2.98. 27

[82] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: a right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 301–315, USA, 2015. USENIX Association. ISBN 9781931971201. 10, 19

[83] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), October 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592438. URL https://doi.org/10.1145/1592434.1592438. 5

[84] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11): 558–562, November 1962. ISSN 0001-0782. doi: 10.1145/368996.369025. URL https://doi.org/10.1145/368996.369025. 29

[85] Richard M. KARP and Vijaya RAMACHANDRAN. Chapter 17 - parallel algorithms for shared-memory machines. In JAN VAN LEEUWEN, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 869–941. Elsevier, Amsterdam, 1990. ISBN 978-0-444-88071-0. doi: https://doi.org/10.1016/B978-0-444-88071-0.50022-9. URL https://www.sciencedirect.com/science/article/pii/B9780444880710500229. 31

[86] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography: Principles and Protocols*. Chapman and Hall/CRC, New York, 1st edition, 2007. ISBN 9780429143809. doi: 10.1201/9781420010756. URL https://doi.org/10.1201/9781420010756. 5

[87] J. Mark Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14(4):799–817, 1985. doi: 10.1137/0214056. 43

[88] David Kirkpatrick and Jack Snoeyink. Computing common tangents without a separating line. In Selim G. Akl, Frank Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, pages 183–193, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-44747-4. doi: 10.1007/3-540-60220-8_61. 39

[89] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201896850. 10

[90] C. Kolovson and M. Stonebraker. Indexing techniques for historical databases. In *[1989] Proceedings. Fifth International Conference on Data Engineering*, pages 127–137, 1989. doi: 10.1109/ICDE.1989.47208. 19

[91] V. Kumar and E.J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–176, 1996. doi: 10.1109/SPDP.1996.570330. 12

[92] Sitaram Lanka and Eric Mays. Fully persistent b+-trees. *SIGMOD Rec.*, 20(2): 426–435, April 1991. ISSN 0163-5808. doi: 10.1145/119995.115861. 23

[93] Aldo Laurentini. Guarding the walls of an art gallery. *The Visual Computer*, 15(6):265–278, October 1999. ISSN 0178-2789. doi: 10.1007/s003710050177. URL `https://doi.org/10.1007/s003710050177`. 42

[94] Craig A. Layman, D. Albrey Arrington, Carmen G. Montaña, and David M. Post. Can stable isotope ratios provide for community-wide measures of trophic structure? *Ecology*, 88(1):42–48, 2007. doi: https://doi.org/10.1890/0012-9658(2007)88[42:CSIRPF]2.0.CO;2. 41

[95] C.C. Lee and D.T. Lee. On a circle-cover minimization problem. *Information Processing Letters*, 18(2):109–115, 1984. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(84)90033-4. URL `https://www.sciencedirect.com/science/article/pii/0020019084900334`. 43, 44

[96] D. T. Lee and Arthur K. Lin. *Computational Complexity of Art Gallery Problems*, pages 303–309. Springer New York, New York, NY, 1990. ISBN 978-1-4613-8997-2. doi: 10.1007/978-1-4613-8997-2_23. URL `https://doi.org/10.1007/978-1-4613-8997-2_23`. 42

[97] Leonid A Levin. Universal sequential search problems. *Problems of information transmission*, 9(3):265–266, 1973. 5

[98] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008. doi: 10.1109/ICDM.2008.17. 38, 41

[99] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. *ACM Trans. Knowl. Discov. Data*, 6(1), March 2012. ISSN 1556-4681. doi: 10.1145/2133360.2133363. URL `https://doi.org/10.1145/2133360.2133363`. 41

[100] Maarten Löffler and Wolfgang Mulzer. Unions of onions: preprocessing imprecise points for fast onion decomposition. *Journal of Computational Geometry*, 5(1), 2014. doi: 10.20382/jocg.v5i1a1. 39, 41

[101] N. Malini and M. Pushpa. Analysis on credit card fraud identification techniques based on knn and outlier detection. In *2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, pages 255–258, 2017. doi: 10.1109/AEEICB.2017.7972424. 4

[102] Jiří Matoušek. Intersection graphs of segments and $\exists\mathbb{R}$. arXiv preprint arXiv:1406.2636, 2014. 42

[103] K. L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45069-6. 5

[104] Magnus C. R. Merrild, Casper M. Rysgaard, Jens K. R. Schou, and Rolf Svenning. An Algorithm for the Contiguous Art Gallery Problem in C++ . `https://github.com/RolfSvenning/ContiguousArtGallery`, 2024. 42

[105] J. Ian Munro and Yakov Nekrich. Dynamic Planar Point Location in External Memory. In Gill Barequet and Yusu Wang, editors, *35th International Symposium on Computational Geometry (SoCG 2019)*, volume 129 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-104-7. doi: 10.4230/LIPIcs.SoCG.2019.52. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SoCG.2019.52`. 25

[106] Ray Bradford Murphy. *On Tests for Outlying Observations*. Phd thesis, Princeton University, 1951. 38

[107] Edgar Acu na and Caroline Rodríguez. A meta analysis study of outlier detection methods in classification. Technical report, Department of Mathematics, University of Puerto Rico at Mayagüez, 2004. 38

[108] Johan Nordlander, Magnus Carlsson, and Andy J. Gill. Unrestricted pure call-by-value recursion. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, page 23–34, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580623. doi: 10.1145/1411304.1411309. URL `https://doi.org/10.1145/1411304.1411309`. 27

[109] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, USA, 1999. ISBN 0521663504. 26, 27

[110] Joseph O'Rourke. *Art gallery theorems and algorithms*. Oxford University Press, Inc., USA, 1987. ISBN 0195039653. 41, 43

[111] Mark H. Overmars. Searching in the past II: general transforms. Technical report, Tech. Rep. RUU, 1981. 17

[112] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. ISBN 3-540-12330-X. doi: 10.1007/BFb0014927. 13, 17, 20

[113] Mark H. Overmars and Jan van Leeuwen. Dynamization of decomposable searching problems yielding good worst-case bounds. In Peter Deussen, editor, *Theoretical Computer Science*, pages 224–233, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg. ISBN 978-3-540-38561-5. 20

[114] Mark H Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of computer and System Sciences*, 23(2):166–204, 1981. 39

[115] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957. doi: 10.1002/j.1538-7305.1957.tb01515.x. 11

[116] Mathias Rav, Aaron Lowe, and Pankaj K. Agarwal. Flood risk analysis on terrains. *ACM Trans. Spatial Algorithms Syst.*, 5(1), June 2019. ISSN 2374-0353. doi: 10.1145/3295459. URL https://doi.org/10.1145/3295459. 10

[117] Eliot W. Robson, Jack Spalding-Jamieson, and Da Wei Zheng. The analytic arc cover problem and its applications to contiguous art gallery, polygon separation, and shape carving, 2025. URL https://arxiv.org/abs/2412.15567. 7, 42

[118] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986. ISSN 0001-0782. doi: 10.1145/6138.6151. URL https://doi.org/10.1145/6138.6151. 28

[119] Marcus Schaefer and Daniel Štefankovič. Fixed points, nash equilibria, and the existential theory of the reals. *Theory of Computing Systems*, 60(2):172–193, 2017. doi: 10.1007/s00224-015-9662-0. URL https://doi.org/10.1007/s00224-015-9662-0. 42

[120] Jacob T. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, October 1980. ISSN 0164-0925. doi: 10.1145/357114.357116. URL https://doi.org/10.1145/357114.357116. 10

[121] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet,

J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 68658–68685. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/7ede97c3e082c6df10a8d6103a2eebd2-Paper-Conference.pdf. 5, 10

[122] Michael Ian Shamos. *Computational Geometry*. PhD thesis, Yale University, 1978. 37, 41

[123] Julian Shun and Guy E. Blelloch. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Trans. Parallel Comput.*, 1(1), October 2014. ISSN 2329-4949. doi: 10.1145/2661653. URL https://doi.org/10.1145/2661653. 32, 34

[124] Julian Shun and Fuyao Zhao. Practical parallel lempel-ziv factorization. In *2013 Data Compression Conference*, pages 123–132, 2013. doi: 10.1109/DCC.2013.20. 32, 34

[125] Nodari Sitchinava and Rolf Svenning. The all nearest smaller values problem revisited in practice, parallel and external memory. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '24, page 259–268, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704161. doi: 10.1145/3626183.3659979. URL https://doi.org/10.1145/3626183.3659979. 7

[126] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, February 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL https://doi.org/10.1145/2786.2793. 11

[127] Rolf Svenning and Vinesh Sridhar. Fast area-weighted peeling of convex hulls for outlier detection. In *Proceedings of the 36th Canadian Conference on Computational Geometry*, pages 233–240. The CCCG Library, July 2024. 36th Canadian Conference on Computational Geometry, CCCG 2024 ; Conference date: 17-07-2024 Through 19-07-2024. 6, 38

[128] Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(79)90042-4. URL https://www.sciencedirect.com/science/article/pii/0022000079900424. 27

[129] Mikkel Thorup. On ram priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, page 59–67, USA, 1996. Society for Industrial and Applied Mathematics. ISBN 0898713668. 12

[130] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363), 1936. 5, 7, 42

[131] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: reference counting optimized for purely functional programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, IFL '19, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450375627. doi: 10.1145/3412932.3412935. URL https://doi.org/10.1145/3412932.3412935. 27

[132] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL https://doi.org/10.1145/79173.79181. 31

[133] Melissa van Beekveld, Sascha Caron, Luc Hendriks, Paul Jackson, Adam Leinweber, Sydney Otten, Riley Patrick, Roberto Ruiz de Austri, Marco Santoni, and Martin White. Combining outlier analysis algorithms to identify new physics at the lhc. *Journal of High Energy Physics*, 2021, 2021. ISSN 1029-8479. doi: 10.1007/JHEP09(2021)024. 4

[134] Jan van Leeuwen and Derick Wood. The measure problem for rectangular ranges in d-space. *Journal of Algorithms*, 2(3):282–300, 1981. ISSN 0196-6774. doi: https://doi.org/10.1016/0196-6774(81)90027-4. URL https://www.sciencedirect.com/science/article/pii/0196677481900274. 26

[135] Vijay V Vazirani. *Approximation algorithms*, volume 1. Springer, 2001. 5

[136] Jeffrey Scott Vitter et al. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008. 23

[137] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, April 1978. ISSN 0001-0782. doi: 10.1145/359460.359478. URL https://doi.org/10.1145/359460.359478. 12, 13

[138] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '77, page 214–222, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373555. doi: 10.1145/563858.563896. 44

[139] John William Joseph Williams. Algorithm 232: heapsort. *Communications of the ACM*, 7(6):347–348, 1964. 11, 14, 39

[140] James C. Wyllie. The complexity of parallel computations. Technical report, Department of Computer Science, Cornell University, 1979. 31